Godot Docs – *4.3* branch

Note

Godot's documentation is available in various languages and versions. Expand the "Read the Docs" panel at the bottom of the sidebar to see the list.

Welcome to the official documentation of Godot Engine [https://godotengine.org], the free and open source community-driven 2D and 3D game engine! If you are new to this documentation, we recommend that you read the introduction page to get an overview of what this documentation has to offer.

The table of contents in the sidebar should let you easily access the documentation for your topic of interest. You can also use the search function in the top-left corner.

Get involved

Godot Engine is an open source project developed by a community of volunteers. The documentation team can always use your feedback and help to improve the tutorials and class reference. If you don't understand something, or cannot find what you are looking for in the docs, help us make the documentation better by letting us know!

Submit an issue or pull request on the GitHub repository [https://github.com/godotengine/godot-docs/issues], help us translate the documentation [https://hosted.weblate.org/engage/godot-engine/] into your language, or talk to us on the `#documentation` channel on the Godot Contributors Chat [https://chat.godotengine.org/]!

25,414 STRINGS | 58 LANGUAGES | 16% TRANSLATED

[https://hosted.weblate.org/engage/godot-engine/?utm_source=widget]

Offline documentation

To browse the documentation offline, you can download an HTML copy (updated every Monday): stable [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-html-stable.zip], latest [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-html-master.zip], 3.6 [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-html-3.6.zip]. Extract the ZIP archive then open the top-level `index.html` in a web browser.

For mobile devices or e-readers, you can also download an ePub copy (updated every Monday): stable [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-epub-stable.zip], latest [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-epub-master.zip], 3.6 [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-epub-3.6.zip]. Extract the ZIP archive then open the `GodotEngine.epub` file in an e-book reader application.

Godot Docs – *4.3* branch

Note

Godot's documentation is available in various languages and versions. Expand the "Read the Docs" panel at the bottom of the sidebar to see the list.

Welcome to the official documentation of Godot Engine [https://godotengine.org], the free and open source community-driven 2D and 3D game engine! If you are new to this documentation, we recommend that you read the introduction page to get an overview of what this documentation has to offer.

The table of contents in the sidebar should let you easily access the documentation for your topic of interest. You can also use the search function in the top-left corner.

Get involved

Godot Engine is an open source project developed by a community of volunteers. The documentation team can always use your feedback and help to improve the tutorials and class reference. If you don't understand something, or cannot find what you are looking for in the docs, help us make the documentation better by letting us know!

Submit an issue or pull request on the GitHub repository [https://github.com/godotengine/godot-docs/issues], help us translate the documentation [https://hosted.weblate.org/engage/godot-engine/] into your language, or talk to us on the `#documentation` channel on the Godot Contributors Chat [https://chat.godotengine.org/]!

25,414 STRINGS | 58 LANGUAGES | 16% TRANSLATED

[https://hosted.weblate.org/engage/godot-engine/?utm_source=widget]

Offline documentation

To browse the documentation offline, you can download an HTML copy (updated every Monday): stable [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-html-stable.zip], latest [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-html-master.zip], 3.6 [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-html-3.6.zip]. Extract the ZIP archive then open the top-level `index.html` in a web browser.

For mobile devices or e-readers, you can also download an ePub copy (updated every Monday): stable [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-epub-stable.zip], latest [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-epub-master.zip], 3.6 [https://nightly.link/godotengine/godot-docs/workflows/build_offline_docs/master/godot-docs-epub-3.6.zip]. Extract the ZIP archive then open the `GodotEngine.epub` file in an e-book reader application.

Introduction

```
func _ready():
    print("Hello world!")
```

Welcome to the official documentation of **Godot Engine**, the free and open source community-driven 2D and 3D game engine! Behind this mouthful, you will find a powerful yet user-friendly tool that you can use to develop any kind of game, for any platform and with no usage restriction whatsoever.

This page gives a broad overview of the engine and of this documentation, so that you know where to start if you are a beginner or where to look if you need information on a specific feature.

Before you start

The Tutorials and resources page lists video tutorials contributed by the community. If you prefer video to text, consider checking them out. Otherwise, Getting Started is a great starting point.

In case you have trouble with one of the tutorials or your project, you can find help on the various Community channels [https://godotengine.org/community/], especially the Godot Discord [https://discord.gg/bdcfAYM4W9] community and Forum [https://forum.godotengine.org/].

About Godot Engine

A game engine is a complex tool and difficult to present in a few words. Here's a quick synopsis, which you are free to reuse if you need a quick write-up about Godot Engine:

> Godot Engine is a feature-packed, cross-platform game engine to create 2D and 3D games from a unified interface. It provides a comprehensive set of common tools, so that users can focus on making games without having to reinvent the wheel. Games can be exported with one click to a number of platforms, including the major desktop platforms (Linux, macOS, Windows), mobile platforms (Android, iOS), as well as Web-based platforms and consoles.
>
> Godot is completely free and open source under the permissive MIT license. No strings attached, no royalties, nothing. Users' games are theirs, down to the last line of engine code. Godot's development is fully independent and community-driven, empowering users to help shape their engine to match their expectations. It is supported by the Godot Foundation [https://godot.foundation/] not-for-profit.

Organization of the documentation

This documentation is organized into several sections:

- **About** contains this introduction as well as information about the engine, its history, its licensing, authors, etc. It also contains the Frequently asked questions.
- **Getting Started** contains all necessary information on using the engine to make games. It starts with the Step by step tutorial which should be the entry point for all new users. **This is the best place to start if you're new!**
- The **Manual** can be read or referenced as needed, in any order. It contains feature-specific tutorials and documentation.
- **Contributing** gives information related to contributing to Godot, whether to the core engine, documentation, demos or other parts. It describes how to report bugs, how contributor

workflows are organized, etc. It also contains sections intended for advanced users and contributors, with information on compiling the engine, contributing to the editor, or developing C++ modules.

- **Community** is dedicated to the life of Godot's community and contains a list of recommended third-party tutorials and materials outside of this documentation. It also provides details on the Asset Library. It also used to list Godot communities, which are now listed on the Godot website [https://godotengine.org/community/].
- Finally, the **Class reference** documents the full Godot API, also available directly within the engine's script editor. You can find information on all classes, functions, signals and so on here.

In addition to this documentation, you may also want to take a look at the various Godot demo projects [https://github.com/godotengine/godot-demo-projects].

About this documentation

Members of the Godot Engine community continuously write, correct, edit, and improve this documentation. We are always looking for more help. You can also contribute by opening Github issues or translating the documentation into your language. If you are interested in helping, see Ways to contribute and Writing documentation, or get in touch with the Documentation team [https://godotengine.org/teams/#documentation] on Godot Contributors Chat [https://chat.godotengine.org/].

All documentation content is licensed under the permissive Creative Commons Attribution 3.0 (CC BY 3.0 [https://creativecommons.org/licenses/by/3.0/]) license, with attribution to "*Juan Linietsky, Ariel Manzur, and the Godot Engine community*" unless otherwise noted.

Have fun reading and making games with Godot Engine!

List of features

This page aims to list **all** features currently supported by Godot.

Note

This page lists features supported by the current stable version of Godot. Some of these features may not be available in the LTS release series (3.x) [https://docs.godotengine.org/en/3.5/about/list_of_features.html].

Platforms

See also

See System requirements for hardware and software version requirements.

Can run both the editor and exported projects:

- Windows (x86 and ARM, 64-bit and 32-bit).
- macOS (x86 and ARM, 64-bit only).
- Linux (x86 and ARM, 64-bit and 32-bit).
    - Binaries are statically linked and can run on any distribution if compiled on an old enough base distribution.
    - Official binaries are compiled using the Godot Engine buildroot [https://github.com/godotengine/buildroot], allowing

for binaries that work across common Linux distributions (including LTS variants).

- Android (editor support is experimental).
- Web browsers. Experimental in 4.0, using Godot 3.x is recommended instead when targeting HTML5.

Runs exported projects:

- iOS.
- Consoles.

Godot aims to be as platform-independent as possible and can be ported to new platforms with relative ease.

Note

Projects written in C# using Godot 4 currently cannot be exported to the web platform. To use C# on that platform, consider Godot 3 instead. Android and iOS platform support is available as of Godot 4.2, but is experimental and some limitations apply.

Editor

Features:

- Scene tree editor.
- Built-in script editor.
- Support for external script editors such as Visual Studio Code or Vim.
- GDScript debugger.

    - Support for debugging in threads is available since 4.2.
- Visual profiler with CPU and GPU time indications for each step of the rendering pipeline.
- Performance monitoring tools, including custom performance monitors.
- Live script reloading.
- Live scene editing.
    - Changes will reflect in the editor and will be kept after closing the running project.
- Remote inspector.
    - Changes won't reflect in the editor and won't be kept after closing the running project.
- Live camera replication.
    - Move the in-editor camera and see the result in the running project.
- Built-in offline class reference documentation.
- Use the editor in dozens of languages contributed by the community.

Plugins:

- Editor plugins can be downloaded from the asset library to extend editor functionality.
- Create your own plugins using GDScript to add new features or speed up your workflow.
- Download projects from the asset library in the Project Manager and import them directly.

Rendering

3 rendering *methods* (running over 2 rendering *drivers*) are available:

- **Forward+**, running over Vulkan 1.0 (with optional Vulkan 1.1 and 1.2 features). The most advanced graphics backend, suited for desktop platforms only. Used by default on desktop platforms.
- **Forward Mobile**, running over Vulkan 1.0 (with optional Vulkan 1.1 and 1.2 features). Less features, but renders simple scenes faster. Suited for mobile and desktop platforms. Used by default on mobile platforms.
- **Compatibility**, running over OpenGL 3.3 / OpenGL ES 3.0 / WebGL 2.0. The least advanced graphics backend, suited for low-end desktop and mobile platforms. Used by default on the web platform.

2D graphics

- Sprite, polygon and line rendering.
    - High-level tools to draw lines and polygons such as Polygon2D and Line2D, with support for texturing.
- AnimatedSprite2D as a helper for creating animated sprites.
- Parallax layers.
    - Pseudo-3D support including preview in the editor.
- 2D lighting with normal maps and specular maps.
    - Point (omni/spot) and directional 2D lights.
    - Hard or soft shadows (adjustable on a per-light basis).

    - Custom shaders can access a real-time SDF representation of the 2D scene based on LightOccluder2D nodes, which can be used for improved 2D lighting effects including 2D global illumination.

- Font rendering using bitmaps, rasterization using FreeType or multi-channel signed distance fields (MSDF).

    - Bitmap fonts can be exported using tools like BMFont, or imported from images (for fixed-width fonts only).
    - Dynamic fonts support monochrome fonts as well as colored fonts (e.g. for emoji). Supported formats are TTF, OTF, WOFF1 and WOFF2.
    - Dynamic fonts support optional font outlines with adjustable width and color.
    - Dynamic fonts support variable fonts and OpenType features including ligatures.
    - Dynamic fonts support simulated bold and italic when the font file lacks those styles.
    - Dynamic fonts support oversampling to keep fonts sharp at higher resolutions.
    - Dynamic fonts support subpixel positioning to make fonts crisper at low sizes.
    - Dynamic fonts support LCD subpixel optimizations to make fonts even crisper at low sizes.
    - Signed distance field fonts can be scaled at any resolution without requiring re-rasterization. Multi-channel usage makes SDF fonts scale down to lower sizes better compared to monochrome SDF fonts.

- GPU-based particles with support for custom particle shaders.

- CPU-based particles.

- Optional 2D HDR rendering for better glow capabilities.

2D tools

- TileMaps for 2D tile-based level design.

- 2D camera with built-in smoothing and drag margins.

- Path2D node to represent a path in 2D space.

    - Can be drawn in the editor or generated procedurally.
    - PathFollow2D node to make nodes follow a Path2D.

- 2D geometry helper class.

2D physics

Physics bodies:

- Static bodies.
- Animatable bodies (for objects moving only by script or animation, such as doors and platforms).
- Rigid bodies.
- Character bodies.
- Joints.
- Areas to detect bodies entering or leaving it.

Collision detection:

- Built-in shapes: line, box, circle, capsule, world boundary (infinite plane).
- Collision polygons (can be drawn manually or generated from a sprite in the editor).

3D graphics

- HDR rendering with sRGB.
- Perspective, orthographic and frustum-offset cameras.
- When using the Forward+ backend, a depth prepass is used to improve performance in complex scenes by reducing the cost of overdraw.
- Variable rate shading on supported GPUs in Forward+ and Forward Mobile.

Physically-based rendering (built-in material features):

- Follows the Disney PBR model.
- Supports Burley, Lambert, Lambert Wrap (half-Lambert) and Toon diffuse shading modes.
- Supports Schlick-GGX, Toon and Disabled specular shading modes.
- Uses a roughness-metallic workflow with support for ORM textures.
- Uses horizon specular occlusion (Filament model) to improve material appearance.
- Normal mapping.
- Parallax/relief mapping with automatic level of detail based on distance.
- Detail mapping for the albedo and normal maps.
- Sub-surface scattering and transmittance.
- Screen-space refraction with support for material roughness (resulting in blurry refraction).
- Proximity fade (soft particles) and distance fade.
- Distance fade can use alpha blending or dithering to avoid going through the transparent pipeline.
- Dithering can be determined on a per-pixel or per-object basis.

Real-time lighting:

- Directional lights (sun/moon). Up to 4 per scene.
- Omnidirectional lights.
- Spot lights with adjustable cone angle and attenuation.

- Specular, indirect light, and volumetric fog energy can be adjusted on a per-light basis.
- Adjustable light "size" for fake area lights (will also make shadows blurrier).
- Optional distance fade system to fade distant lights and their shadows, improving performance.
- When using the Forward+ backend (default on desktop), lights are rendered with clustered forward optimizations to decrease their individual cost. Clustered rendering also lifts any limits on the number of lights that can be used on a mesh.
- When using the Forward Mobile backend, up to 8 omni lights and 8 spot lights can be displayed per mesh resource. Baked lighting can be used to overcome this limit if needed.

Shadow mapping:

- *DirectionalLight:* Orthogonal (fastest), PSSM 2-split and 4-split. Supports blending between splits.
- *OmniLight:* Dual paraboloid (fast) or cubemap (slower but more accurate). Supports colored projector textures in the form of panoramas.
- *SpotLight:* Single texture. Supports colored projector textures.
- Shadow normal offset bias and shadow pancaking to decrease the amount of visible shadow acne and peter-panning.
- PCSS-like shadow blur based on the light size and distance from the surface the shadow is cast on.
- Adjustable shadow blur on a per-light basis.

Global illumination with indirect lighting:

- Baked lightmaps (fast, but can't be updated at run-time).
  - Supports baking indirect light only or baking both direct and indirect lighting. The bake mode can be adjusted on a per-light basis to allow for hybrid light baking setups.

  - Supports lighting dynamic objects using automatic and manually placed probes.
  - Optionally supports directional lighting and rough reflections based on spherical harmonics.
  - Lightmaps are baked on the GPU using compute shaders (much faster compared to CPU lightmapping). Baking can only be performed from the editor, not in exported projects.
  - Supports GPU-based denoising with JNLM, or CPU/GPU-based denoising with OIDN.

- Voxel-based GI probes. Supports dynamic lights *and* dynamic occluders, while also supporting reflections. Requires a fast baking step which can be performed in the editor or at run-time (including from an exported project).

- Signed-distance field GI designed for large open worlds. Supports dynamic lights, but not dynamic occluders. Supports reflections. No baking required.

- Screen-space indirect lighting (SSIL) at half or full resolution. Fully real-time and supports any kind of emissive light source (including decals).

- VoxelGI and SDFGI use a deferred pass to allow for rendering GI at half resolution to improve performance (while still having functional MSAA support).

Reflections:

- Voxel-based reflections (when using GI probes) and SDF-based reflections (when using signed distance field GI). Voxel-based reflections are visible on transparent surfaces, while rough SDF-based reflections are visible on transparent surfaces.
- Fast baked reflections or slow real-time reflections using ReflectionProbe. Parallax box correction can optionally be enabled.

- Screen-space reflections with support for material roughness.
- Reflection techniques can be mixed together for greater accuracy or scalability.
- When using the Forward+ backend (default on desktop), reflection probes are rendered with clustered forward optimizations to decrease their individual cost. Clustered rendering also lifts any limits on the number of reflection probes that can be used on a mesh.
- When using the Forward Mobile backend, up to 8 reflection probes can be displayed per mesh resource.

Decals:

- Supports albedo, emissive, ORM, and normal mapping.
- Texture channels are smoothly overlaid on top of the underlying material, with support for normal/ORM-only decals.
- Support for normal fade to fade the decal depending on its incidence angle.
- Does not rely on run-time mesh generation. This means decals can be used on complex skinned meshes with no performance penalty, even if the decal moves every frame.
- Support for nearest, bilinear, trilinear or anisotropic texture filtering (configured globally).
- Optional distance fade system to fade distant decals, improving performance.
- When using the Forward+ backend (default on desktop), decals are rendered with clustered forward optimizations to decrease their individual cost. Clustered rendering also lifts any limits on the number of decals that can be used on a mesh.
- When using the Forward Mobile backend, up to 8 decals can be displayed per mesh resource.

Sky:

- Panorama sky (using an HDRI).

- Procedural sky and Physically-based sky that respond to the DirectionalLights in the scene.
- Support for custom sky shaders, which can be animated.
- The radiance map used for ambient and specular light can be updated in real-time depending on the quality settings chosen.

Fog:

- Exponential depth fog.
- Exponential height fog.
- Support for automatic fog color depending on the sky color (aerial perspective).
- Support for sun scattering in the fog.
- Support for controlling how much fog rendering should affect the sky, with separate controls for traditional and volumetric fog.
- Support for making specific materials ignore fog.

Volumetric fog:

- Global volumetric fog that reacts to lights and shadows.
- Volumetric fog can take indirect light into account when using VoxelGI or SDFGI.
- Fog volume nodes that can be placed to add fog to specific areas (or remove fog from specific areas). Supported shapes include box, ellipse, cone, cylinder, and 3D texture-based density maps.
- Each fog volume can have its own custom shader.
- Can be used together with traditional fog.

Particles:

- GPU-based particles with support for subemitters (2D + 3D), trails (2D + 3D), attractors (3D only) and collision (2D + 3D).
  - 3D particle attractor shapes supported: box, sphere and 3D vector fields.
  - 3D particle collision shapes supported: box, sphere, baked signed distance field and real-time heightmap (suited for

    open world weather effects).
    - 2D particle collision is handled using a signed distance field generated in real-time based on LightOccluder2D nodes in the scene.
    - Trails can use the built-in ribbon trail and tube trail meshes, or custom meshes with skeletons.
    - Support for custom particle shaders with manual emission.
- CPU-based particles.

Post-processing:

- Tonemapping (Linear, Reinhard, Filmic, ACES).
- Automatic exposure adjustments based on viewport brightness (and manual exposure override).
- Near and far depth of field with adjustable bokeh simulation (box, hexagon, circle).
- Screen-space ambient occlusion (SSAO) at half or full resolution.
- Glow/bloom with optional bicubic upscaling and several blend modes available: Screen, Soft Light, Add, Replace, Mix.
- Glow can have a colored dirt map texture, acting as a lens dirt effect.
- Glow can be used as a screen-space blur effect.
- Color correction using a one-dimensional ramp or a 3D LUT texture.
- Roughness limiter to reduce the impact of specular aliasing.
- Brightness, contrast and saturation adjustments.

Texture filtering:

- Nearest, bilinear, trilinear or anisotropic filtering.
- Filtering options are defined on a per-use basis, not a per-texture basis.

Texture compression:

- Basis Universal (slow, but results in smaller files).
- BPTC for high-quality compression (not supported on macOS).

- ETC2 (not supported on macOS).
- S3TC (not supported on mobile/Web platforms).

Anti-aliasing:

- Temporal antialiasing (TAA).
- AMD FidelityFX Super Resolution 2.2 antialiasing (FSR2), which can be used at native resolution as a form of high-quality temporal antialiasing.
- Multi-sample antialiasing (MSAA), for both 2D antialiasing and 3D antialiasing.
- Fast approximate antialiasing (FXAA).
- Super-sample antialiasing (SSAA) using bilinear 3D scaling and a 3D resolution scale above 1.0.
- Alpha antialiasing, MSAA alpha to coverage and alpha hashing on a per-material basis.

Resolution scaling:

- Support for rendering 3D at a lower resolution while keeping 2D rendering at the original scale. This can be used to improve performance on low-end systems or improve visuals on high-end systems.
- Resolution scaling uses bilinear filtering, AMD FidelityFX Super Resolution 1.0 (FSR1) or AMD FidelityFX Super Resolution 2.2 (FSR2).
- Texture mipmap LOD bias is adjusted automatically to improve quality at lower resolution scales. It can also be modified with a manual offset.

Most effects listed above can be adjusted for better performance or to further improve quality. This can be helpful when using Godot for offline rendering.

3D tools

- Built-in meshes: cube, cylinder/cone, (hemi)sphere, prism, plane, quad, torus, ribbon, tube.

- GridMaps for 3D tile-based level design.

- Constructive solid geometry (intended for prototyping).

- Tools for procedural geometry generation.

- Path3D node to represent a path in 3D space.

    - Can be drawn in the editor or generated procedurally.
    - PathFollow3D node to make nodes follow a Path3D.

- 3D geometry helper class.

- Support for exporting the current scene as a glTF 2.0 file, both from the editor and at run-time from an exported project.

3D physics

Physics bodies:

- Static bodies.
- Animatable bodies (for objects moving only by script or animation, such as doors and platforms).
- Rigid bodies.
- Character bodies.
- Vehicle bodies (intended for arcade physics, not simulation).
- Joints.
- Soft bodies.
- Ragdolls.
- Areas to detect bodies entering or leaving it.

Collision detection:

- Built-in shapes: cuboid, sphere, capsule, cylinder, world boundary (infinite plane).
- Generate triangle collision shapes for any mesh from the editor.
- Generate one or several convex collision shapes for any mesh from the editor.

Shaders

- *2D:* Custom vertex, fragment, and light shaders.

- *3D:* Custom vertex, fragment, light, and sky shaders.

- Text-based shaders using a shader language inspired by GLSL.

- Visual shader editor.

    - Support for visual shader plugins.

Scripting

General:

- Object-oriented design pattern with scripts extending nodes.
- Signals and groups for communicating between scripts.
- Support for cross-language scripting.
- Many 2D, 3D and 4D linear algebra data types such as vectors and transforms.

GDScript:

- High-level interpreted language with optional static typing.
- Syntax inspired by Python. However, GDScript is **not** based on Python.
- Syntax highlighting is provided on GitHub.

- Use threads to perform asynchronous actions or make use of multiple processor cores.

C#:

- Packaged in a separate binary to keep file sizes and dependencies down.
- Supports .NET 6 and higher.
  - Full support for the C# 10.0 syntax and features.
- Supports Windows, Linux, and macOS. As of 4.2 experimental support for Android and iOS is also available (requires a .NET 7.0 project for Android and 8.0 for iOS).
  - On the Android platform only some architectures are supported: `arm64` and `x64`.
  - On the iOS platform only some architectures are supported: `arm64`.
  - The web platform is currently unsupported. To use C# on that platform, consider Godot 3 instead.
- Using an external editor is recommended to benefit from IDE functionality.

GDExtension (C, C++, Rust, D, ...):

- When you need it, link to native libraries for higher performance and third-party integrations.
  - For scripting game logic, GDScript or C# are recommended if their performance is suitable.
- Official GDExtension bindings for C [https://github.com/godotengine/godot-headers] and C++ [https://github.com/godotengine/godot-cpp].

    - Use any build system and language features you wish.

- Actively developed GDExtension bindings for D [https://github.com/godot-dlang/godot-dlang], Haxe [https://hxgodot.github.io/], Swift [https://github.com/migueldeicaza/SwiftGodot], and Rust [https://github.com/godot-rust/gdextension] bindings provided by the community. (Some of these bindings may be experimental and not production-ready).

Audio

Features:

- Mono, stereo, 5.1 and 7.1 output.
- Non-positional and positional playback in 2D and 3D.
    - Optional Doppler effect in 2D and 3D.
- Support for re-routable audio buses and effects with dozens of effects included.
- Support for polyphony (playing several sounds from a single AudioStreamPlayer node).
- Support for random volume and pitch.
- Support for real-time pitch scaling.
- Support for sequential/random sample selection, including repetition prevention when using random sample selection.
- Listener2D and Listener3D nodes to listen from a position different than the camera.

- Support for procedural audio generation.
- Audio input to record microphones.
- MIDI input.
    - No support for MIDI output yet.

APIs used:

- *Windows:* WASAPI.
- *macOS:* CoreAudio.
- *Linux:* PulseAudio or ALSA.

Import

- Support for custom import plugins.

Formats:

- *Images:* See Importing images.
- *Audio:*
    - WAV with optional IMA-ADPCM compression.
    - Ogg Vorbis.
    - MP3.
- *3D scenes:* See Importing 3D scenes.
    - glTF 2.0 *(recommended)*.
    - `.blend` (by calling Blender's glTF export functionality transparently).
    - FBX (by calling FBX2glTF [https://github.com/godotengine/FBX2glTF] transparently).
    - Collada (.dae).

  - Wavefront OBJ (static scenes only, can be loaded directly as a mesh or imported as a 3D scene).

- Support for loading glTF 2.0 scenes at run-time, including from an exported project.

- 3D meshes use Mikktspace [http://www.mikktspace.com/] to generate tangents on import, which ensures consistency with other 3D applications such as Blender.

Input

- Input mapping system using hardcoded input events or remappable input actions.

  - Axis values can be mapped to two different actions with a configurable deadzone.
  - Use the same code to support both keyboards and gamepads.

- Keyboard input.

  - Keys can be mapped in "physical" mode to be independent of the keyboard layout.

- Mouse input.

  - The mouse cursor can be visible, hidden, captured or confined within the window.
  - When captured, raw input will be used on Windows and Linux to sidestep the OS' mouse acceleration settings.

- Gamepad input (up to 8 simultaneous controllers).

- Pen/tablet input with pressure support.

Navigation

- A* algorithm in 2D and 3D.
- Navigation meshes with dynamic obstacle avoidance in 2D and 3D.
- Generate navigation meshes from the editor or at run-time (including from an exported project).

Networking

- Low-level TCP networking using StreamPeer and TCPServer.
- Low-level UDP networking using PacketPeer and UDPServer.
- Low-level HTTP requests using HTTPClient.
- High-level HTTP requests using HTTPRequest.
    - Supports HTTPS out of the box using bundled certificates.
- High-level multiplayer API using UDP and ENet.
    - Automatic replication using remote procedure calls (RPCs).
    - Supports unreliable, reliable and ordered transfers.
- WebSocket client and server, available on all platforms.
- WebRTC client and server, available on all platforms.
- Support for UPnP to sidestep the requirement to forward ports when hosting a server behind a NAT.

Internationalization

- Full support for Unicode including emoji.
- Store localization strings using CSV or gettext.
    - Support for generating gettext POT and PO files from the editor.
- Use localized strings in your project automatically in GUI elements or by using the `tr()` function.
- Support for pluralization and translation contexts when using gettext translations.
- Support for bidirectional typesetting, text shaping and OpenType localized forms.
- Automatic UI mirroring for right-to-left locales.
- Support for pseudolocalization to test your project for i18n-friendliness.

Windowing and OS integration

- Spawn multiple independent windows within a single process.
- Move, resize, minimize, and maximize windows spawned by the project.
- Change the window title and icon.
- Request attention (will cause the title bar to blink on most platforms).
- Fullscreen mode.
    - Uses borderless fullscreen by default on Windows for fast alt-tabbing, but can optionally use exclusive fullscreen to reduce input lag.
- Borderless windows (fullscreen or non-fullscreen).

- Ability to keep a window always on top.
- Global menu integration on macOS.
- Execute commands in a blocking or non-blocking manner (including running multiple instances of the same project).
- Open file paths and URLs using default or custom protocol handlers (if registered on the system).
- Parse custom command line arguments.
- Any Godot binary (editor or exported project) can be used as a headless server by starting it with the `--headless` command line argument. This allows running the engine without a GPU or display server.

Mobile

- In-app purchases on Android and iOS.
- Support for advertisements using third-party modules.

XR support (AR and VR)

- Out of the box support for OpenXR.
    - Including support for popular desktop headsets like the Valve Index, WMR headsets, and Quest over Link.
- Support for Android based headsets using OpenXR through a plugin.
    - Including support for popular stand alone headsets like the Meta Quest 1/2/3 and Pro, Pico 4, Magic Leap 2, and Lynx R1.

- Other devices supported through an XR plugin structure.
- Various advanced toolkits are available that implement common features required by XR applications.

GUI system

Godot's GUI is built using the same Control nodes used to make games in Godot. The editor UI can easily be extended in many ways using add-ons.

Nodes:

- Buttons.
- Checkboxes, check buttons, radio buttons.
- Text entry using LineEdit (single line) and TextEdit (multiple lines). TextEdit also supports code editing features such as displaying line numbers and syntax highlighting.
- Dropdown menus using PopupMenu and OptionButton.
- Scrollbars.
- Labels.
- RichTextLabel for text formatted using BBCode, with support for animated custom effects.
- Trees (can also be used to represent tables).
- Color picker with RGB and HSV modes.
- Controls can be rotated and scaled.

Sizing:

- Anchors to keep GUI elements in a specific corner, edge or centered.
- Containers to place GUI elements automatically following certain rules.
    - Stack layouts.

    - Grid layouts.
    - Flow layouts (similar to autowrapping text).
    - Margin, centered and aspect ratio layouts.
    - Draggable splitter layouts.

- Scale to multiple resolutions using the `canvas_items` or `viewport` stretch modes.

- Support any aspect ratio using anchors and the `expand` stretch aspect.

Theming:

- Built-in theme editor.

    - Generate a theme based on the current editor theme settings.

- Procedural vector-based theming using StyleBoxFlat.

    - Supports rounded/beveled corners, drop shadows, per-border widths and antialiasing.

- Texture-based theming using StyleBoxTexture.

Godot's small distribution size can make it a suitable alternative to frameworks like Electron or Qt.

Animation

- Direct kinematics and inverse kinematics.
- Support for animating any property with customizable interpolation.
- Support for calling methods in animation tracks.
- Support for playing sounds in animation tracks.
- Support for Bézier curves in animation.

File formats

- Scenes and resources can be saved in text-based or binary formats.
    - Text-based formats are human-readable and more friendly to version control.
    - Binary formats are faster to save/load for large scenes/resources.
- Read and write text or binary files using FileAccess.
    - Can optionally be compressed or encrypted.
- Read and write JSON files.
- Read and write INI-style configuration files using ConfigFile.
    - Can (de)serialize any Godot datatype, including Vector2/3, Color, ...
- Read XML files using XMLParser.
- Load and save images, audio/video, fonts and ZIP archives in an exported project without having to go through Godot's import system.
- Pack game data into a PCK file (custom format optimized for fast seeking), into a ZIP archive, or directly into the executable for single-file distribution.
- Export additional PCK files that can be read by the engine to support mods and DLCs.

Miscellaneous

- Video playback with built-in support for Ogg Theora.
- Movie Maker mode to record videos from a running project with synchronized audio and perfect frame pacing.
- Low-level access to servers which allows bypassing the scene tree's overhead when needed.
- Command line interface for automation.
    - Export and deploy projects using continuous integration platforms.
    - Shell completion scripts [https://github.com/godotengine/godot/tree/master/misc/dist/shell] are available for Bash, zsh and fish.
    - Print colored text to standard output on all platforms using print_rich.
- Support for C++ modules statically linked into the engine binary.
- Engine and editor written in C++17.
    - Can be compiled using GCC, Clang and MSVC. MinGW is also supported.
    - Friendly towards packagers. In most cases, system libraries can be used instead of the ones provided by Godot. The build system doesn't download anything. Builds can be fully reproducible.
- Licensed under the permissive MIT license.
    - Open development process with contributions welcome.

See also

The Godot proposals repository [https://github.com/godotengine/godot-proposals] lists features that have been requested by the community and may be implemented in future Godot releases.

System requirements

This page contains system requirements for the editor and exported projects. These specifications are given for informative purposes only, but they can be referred to if you're looking to build or upgrade a system to use Godot on.

Godot editor

These are the **minimum** specifications required to run the Godot editor and work on a simple 2D or 3D project:

Desktop or laptop PC - Minimum

CPU	• **Windows:** x86_32 CPU with SSE2 instructions, x86_64 CPU, ARMv8 CPU ◦ *Example: Intel Core 2 Duo E8200, AMD Athlon XE BE-2300, Snapdragon X Elite* • **macOS:** x86_64 or ARM CPU (Apple Silicon) ◦ *Example: Intel Core 2 Duo SU9400, Apple M1* • **Linux:** x86_32 CPU with SSE2 instructions, x86_64 CPU, ARMv7 or ARMv8 CPU ◦ *Example: Intel Core 2 Duo E8200, AMD Athlon XE BE-2300, Raspberry Pi 4*
GPU	• **Forward+ rendering method:** Integrated graphics with full Vulkan 1.0 support ◦ *Example: Intel HD Graphics 5500 (Broadwell), AMD Radeon R5 Graphics*

	(Kaveri) • **Mobile rendering method:** Integrated graphics with full Vulkan 1.0 support ◦ *Example: Intel HD Graphics 5500 (Broadwell), AMD Radeon R5 Graphics (Kaveri)* • **Compatibility rendering method:** Integrated graphics with full OpenGL 3.3 support ◦ *Example: Intel HD Graphics 2500 (Ivy Bridge), AMD Radeon R5 Graphics (Kaveri)*
RAM	• **Native editor:** 4 GB • **Web editor:** 8 GB
Storage	200 MB (used for the executable, project files and cache). Exporting projects requires downloading export templates separately (1.3 GB after installation).
Operating system	• **Native editor:** Windows 7, macOS 10.13 (Compatibility) or macOS 10.15 (Forward+/Mobile), Linux distribution released after 2016 • **Web editor:** Firefox 79, Chrome 68, Edge 79, Safari 15.2, Opera 64

Note

Windows 7/8/8.1 are supported on a best-effort basis. These versions are not regularly tested and some features may be missing (such as colored print_rich console output). Support for Windows 7/8/8.1 may be removed in a future Godot 4.x release.

Vulkan drivers for these Windows versions are known to have issues with memory leaks. As a result, it's recommended to stick to the Compatibility rendering method when running Godot on a Windows version older than 10.

Mobile device (smartphone/tablet) - Minimum

CPU	• **Android:** SoC with any 32-bit or 64-bit ARM or x86 CPU ◦ *Example: Qualcomm Snapdragon 430, Samsung Exynos 5 Octa 5430* • **iOS:** *Cannot run the editor*
GPU	• **Forward+ rendering method:** SoC featuring GPU with full Vulkan 1.0 support ◦ *Example: Qualcomm Adreno 505, Mali-G71 MP2* • **Mobile rendering method:** SoC featuring GPU with full Vulkan 1.0 support ◦ *Example: Qualcomm Adreno 505, Mali-G71 MP2* • **Compatibility rendering method:** SoC featuring GPU with full OpenGL ES 3.0 support ◦ *Example: Qualcomm Adreno 306, Mali-T628 MP6*
RAM	• **Native editor:** 3 GB • **Web editor:** 6 GB

Storage	200 MB (used for the executable, project files and cache). Exporting projects requires downloading export templates separately (1.3 GB after installation).
Operating system	• **Native editor:** Android 6.0 (Compatibility) or Android 9.0 (Forward+/Mobile) • **Web editor:** Firefox 79, Chrome 88, Edge 79, Safari 15.2, Opera 64, Samsung Internet 15

These are the **recommended** specifications to get a smooth experience with the Godot editor on a simple 2D or 3D project:

Desktop or laptop PC - Recommended

CPU	• **Windows:** x86_64 CPU with SSE4.2 instructions, with 4 physical cores or more, ARMv8 CPU ◦ *Example: Intel Core i5-6600K, AMD Ryzen 5 1600, Snapdragon X Elite* • **macOS:** x86_64 or ARM CPU (Apple Silicon) ◦ *Example: Intel Core i5-8500, Apple M1* • **Linux:** x86_32 CPU with SSE2 instructions, x86_64 CPU, ARMv7 or ARMv8 CPU ◦ *Example: Intel Core i5-6600K, AMD Ryzen 5 1600, Raspberry Pi 5 with overclocking*
GPU	• **Forward+ rendering method:** Dedicated graphics with full Vulkan 1.2 support ◦ *Example: NVIDIA GeForce GTX 1050 (Pascal), AMD Radeon RX 460 (GCN 4.0)*

	• **Mobile rendering method:** Dedicated graphics with full Vulkan 1.2 support ◦ *Example: NVIDIA GeForce GTX 1050 (Pascal), AMD Radeon RX 460 (GCN 4.0)* • **Compatibility rendering method:** Dedicated graphics with full OpenGL 4.6 support ◦ *Example: NVIDIA GeForce GTX 650 (Kepler), AMD Radeon HD 7750 (GCN 1.0)*
RAM	• **Native editor:** 8 GB • **Web editor:** 12 GB
Storage	1.5 GB (used for the executable, project files, all export templates and cache)
Operating system	• **Native editor:** Windows 10, macOS 10.15, Linux distribution released after 2020 • **Web editor:** Latest version of Firefox, Chrome, Edge, Safari, Opera

Mobile device (smartphone/tablet) - Recommended

CPU	• **Android:** SoC with 64-bit ARM or x86 CPU, with 3 "performance" cores or more ◦ *Example: Qualcomm Snapdragon 845, Samsung Exynos 9810* • **iOS:** *Cannot run the editor*

GPU	- **Forward+ rendering method:** SoC featuring GPU with full Vulkan 1.2 support
	  - *Example: Qualcomm Adreno 630, Mali-G72 MP18*
	- **Mobile rendering method:** SoC featuring GPU with full Vulkan 1.2 support
	  - *Example: Qualcomm Adreno 630, Mali-G72 MP18*
	- **Compatibility rendering method:** SoC featuring GPU with full OpenGL ES 3.2 support
	  - *Example: Qualcomm Adreno 630, Mali-G72 MP18*

RAM	- **Native editor:** 6 GB - **Web editor:** 8 GB
Storage	1.5 GB (used for the executable, project files, all export templates and cache)
Operating system	- **Native editor:** Android 9.0 - **Web editor:** Latest version of Firefox, Chrome, Edge, Safari, Opera, Samsung Internet

Exported Godot project

Warning

The requirements below are a baseline for a **simple** 2D or 3D project, with basic scripting and few visual flourishes. CPU, GPU, RAM and storage requirements will heavily vary depending on your project's scope, its rendering method, viewport resolution and graphics settings chosen. Other programs running on the system while the project is running will also compete for resources, including RAM and video RAM.

It is strongly recommended to do your own testing on low-end hardware to make sure your project runs at the desired speed. To provide scalability for low-end hardware, you will also need to introduce a graphics options menu [https://github.com/godotengine/godot-demo-projects/tree/master/3d/graphics_settings] to your project.

These are the **minimum** specifications required to run a simple 2D or 3D project exported with Godot:

Desktop or laptop PC - Minimum

CPU	• **Windows:** x86_32 CPU with SSE2 instructions, any x86_64 CPU, ARMv8 CPU
	  • *Example: Intel Core 2 Duo E8200, AMD Athlon XE BE-2300, Snapdragon X Elite*
	• **macOS:** x86_64 or ARM CPU (Apple Silicon)
	  • *Example: Intel Core 2 Duo SU9400, Apple M1*
	• **Linux:** x86_32 CPU with SSE2 instructions, x86_64 CPU, ARMv7 or ARMv8 CPU
	  • *Example: Intel Core 2 Duo E8200, AMD Athlon XE BE-2300, Raspberry Pi 4*

GPU	• **Forward+ rendering method:** Integrated graphics with full Vulkan 1.0 support ◦ *Example: Intel HD Graphics 5500 (Broadwell), AMD Radeon R5 Graphics (Kaveri)* • **Mobile rendering method:** Integrated graphics with full Vulkan 1.0 support ◦ *Example: Intel HD Graphics 5500 (Broadwell), AMD Radeon R5 Graphics (Kaveri)* • **Compatibility rendering method:** Integrated graphics with full OpenGL 3.3 support ◦ *Example: Intel HD Graphics 2500 (Ivy Bridge), AMD Radeon R5 Graphics (Kaveri)*
RAM	• **For native exports:** 2 GB • **For web exports:** 4 GB
Storage	150 MB (used for the executable, project files and cache)
Operating system	• **For native exports:** Windows 7, macOS 10.13 (Compatibility) or macOS 10.15 (Forward+/Mobile), Linux distribution released after 2016 • **For web exports:** Firefox 79, Chrome 68, Edge 79, Safari 15.2, Opera 64

Note

Windows 7/8/8.1 are supported on a best-effort basis. These versions are not regularly tested and some features may be missing (such as colored print_rich console output). Support for Windows 7/8/8.1 may be removed in a future Godot 4.x release.

Vulkan drivers for these Windows versions are known to have issues with memory leaks. As a result, it's recommended to stick to the Compatibility rendering method when running Godot on a Windows version older than 10.

Mobile device (smartphone/tablet) - Minimum

CPU	• **Android:** SoC with any 32-bit or 64-bit ARM or x86 CPU ◦ *Example: Qualcomm Snapdragon 430, Samsung Exynos 5 Octa 5430* • **iOS:** SoC with any 64-bit ARM CPU ◦ *Example: Apple A7 (iPhone 5S)*
GPU	• **Forward+ rendering method:** SoC featuring GPU with full Vulkan 1.0 support ◦ *Example: Qualcomm Adreno 505, Mali-G71 MP2, Apple A12 (iPhone XR/XS)* • **Mobile rendering method:** SoC featuring GPU with full Vulkan 1.0 support ◦ *Example: Qualcomm Adreno 505, Mali-G71 MP2, Apple A12 (iPhone XR/XS)* • **Compatibility rendering method:** SoC featuring GPU with full OpenGL ES 3.0 support ◦ *Example: Qualcomm Adreno 306, Mali-T628 MP6, Apple A7 (iPhone 5S)*
RAM	• **For native exports:** 1 GB • **For web exports:** 2 GB

Storage	150 MB (used for the executable, project files and cache)
Operating system	• **For native exports:** Android 6.0 (Compatibility) or Android 9.0 (Forward+/Mobile), iOS 12.0 • **For web exports:** Firefox 79, Chrome 88, Edge 79, Safari 15.2, Opera 64, Samsung Internet 15

These are the **recommended** specifications to get a smooth experience with a simple 2D or 3D project exported with Godot:

Desktop or laptop PC - Recommended

CPU	• **Windows:** x86_64 CPU with SSE4.2 instructions, with 4 physical cores or more, ARMv8 CPU
	  ◦ *Example: Intel Core i5-6600K, AMD Ryzen 5 1600, Snapdragon X Elite*
	• **macOS:** x86_64 or ARM CPU (Apple Silicon)
	  ◦ *Example: Intel Core i5-8500, Apple M1*
	• **Linux:** x86_32 CPU with SSE2 instructions, x86_64 CPU, ARMv7 or ARMv8 CPU
	  ◦ *Example: Intel Core i5-6600K, AMD Ryzen 5 1600, Raspberry Pi 5 with overclocking*

GPU	• **Forward+ rendering method:** Dedicated graphics with full Vulkan 1.2 support

	◦ *Example: NVIDIA GeForce GTX 1050 (Pascal), AMD Radeon RX 460 (GCN 4.0)* • **Mobile rendering method:** Dedicated graphics with full Vulkan 1.2 support ◦ *Example: NVIDIA GeForce GTX 1050 (Pascal), AMD Radeon RX 460 (GCN 4.0)* • **Compatibility rendering method:** Dedicated graphics with full OpenGL 4.6 support ◦ *Example: NVIDIA GeForce GTX 650 (Kepler), AMD Radeon HD 7750 (GCN 1.0)*
RAM	• **For native exports:** 4 GB • **For web exports:** 8 GB
Storage	150 MB (used for the executable, project files and cache)
Operating system	• **For native exports:** Windows 10, macOS 10.15, Linux distribution released after 2020 • **For web exports:** Latest version of Firefox, Chrome, Edge, Safari, Opera

Mobile device (smartphone/tablet) - Recommended

CPU	• **Android:** SoC with 64-bit ARM or x86 CPU, with 3 "performance" cores or more ◦ *Example: Qualcomm Snapdragon 845, Samsung Exynos 9810* • **iOS:** SoC with 64-bit ARM CPU ◦ *Example: Apple A14 (iPhone 12)*

GPU	- **Forward+ rendering method:** SoC featuring GPU with full Vulkan 1.2 support
	  - *Example: Qualcomm Adreno 630, Mali-G72 MP18, Apple A14 (iPhone 12)*
	- **Mobile rendering method:** SoC featuring GPU with full Vulkan 1.2 support
	  - *Example: Qualcomm Adreno 630, Mali-G72 MP18, Apple A14 (iPhone 12)*
	- **Compatibility rendering method:** SoC featuring GPU with full OpenGL ES 3.2 support
	  - *Example: Qualcomm Adreno 630, Mali-G72 MP18, Apple A14 (iPhone 12)*

RAM	- **For native exports:** 2 GB - **For web exports:** 4 GB
Storage	150 MB (used for the executable, project files and cache)
Operating system	- **For native exports:** Android 9.0 or iOS 14.1 - **For web exports:** Latest version of Firefox, Chrome, Edge, Safari, Opera, Samsung Internet

Note

Godot doesn't use OpenGL/OpenGL ES extensions introduced after OpenGL 3.3/OpenGL ES 3.0, but GPUs supporting newer OpenGL/OpenGL ES versions generally have fewer driver issues.

Frequently asked questions

What can I do with Godot? How much does it cost? What are the license terms?

Godot is Free and open source Software [https://en.wikipedia.org/wiki/Free_and_open_source_software] available under the OSI-approved [https://opensource.org/licenses/MIT] MIT license. This means it is free as in "free speech" as well as in "free beer."

In short:

- You are free to download and use Godot for any purpose: personal, non-profit, commercial, or otherwise.
- You are free to modify, distribute, redistribute, and remix Godot to your heart's content, for any reason, both non-commercially and commercially.

All the contents of this accompanying documentation are published under the permissive Creative Commons Attribution 3.0 (CC BY 3.0 [https://creativecommons.org/licenses/by/3.0/]) license, with attribution to "Juan Linietsky, Ariel Manzur and the Godot Engine community."

Logos and icons are generally under the same Creative Commons license. Note that some third-party libraries included with Godot's source code may have different licenses.

For full details, look at the COPYRIGHT.txt [https://github.com/godotengine/godot/blob/master/COPYRIGHT.txt] as well as the LICENSE.txt [https://github.com/godotengine/godot/blob/master/LICENSE.txt] and LOGO_LICENSE.txt [https://github.com/godotengine/godot/blob/master/LOGO_LICENSE.txt] files in the Godot repository.

Also, see the license page on the Godot website [https://godotengine.org/license].

Which platforms are supported by Godot?

For the editor:

- Windows
- macOS
- Linux, *BSD
- Android (experimental)
- Web [https://editor.godotengine.org/] (experimental)

For exporting your games:

- Windows
- macOS
- Linux, *BSD
- Android
- iOS
- Web

Both 32- and 64-bit binaries are supported where it makes sense, with 64 being the default. Official macOS builds support Apple Silicon natively as well as x86_64.

Some users also report building and using Godot successfully on ARM-based systems with Linux, like the Raspberry Pi.

The Godot team can't provide an open source console export due to the licensing terms imposed by console manufacturers. Regardless of the engine you use, though, releasing games on consoles is always a lot of work. You can read more about Console support in Godot.

For more on this, see the sections on exporting and compiling Godot yourself.

Note

Godot 3 also had support for Universal Windows Platform (UWP). This platform port was removed in Godot 4 due to lack of maintenance, and it being deprecated by Microsoft. It is still available in the current stable release of Godot 3 for interested users.

Which programming languages are supported in Godot?

The officially supported languages for Godot are GDScript, C#, and C++. See the subcategories for each language in the scripting section.

If you are just starting out with either Godot or game development in general, GDScript is the recommended language to learn and use since it is native to Godot. While scripting languages tend to be less performant than lower-level languages in the long run, for prototyping, developing Minimum Viable Products (MVPs), and focusing on Time-To-Market (TTM), GDScript will provide a fast, friendly, and capable way of developing your games.

Note that C# support is still relatively new, and as such, you may encounter some issues along the way. C# support is also currently missing on the web platform. Our friendly and hard-working development community is always ready to tackle new problems as they arise, but since this is an open source project, we recommend that you first do some due diligence yourself. Searching through discussions on open issues [https://github.com/godotengine/godot/issues?q=is%3Aopen+is%3Aissue+label%3Atopic%3Adotnet] is a great way to start your troubleshooting.

As for new languages, support is possible via third parties with GDExtensions. (See the question about plugins below). Work is currently underway, for example, on unofficial bindings for Godot to Python [https://github.com/touilleMan/godot-python] and Nim [https://github.com/pragmagic/godot-nim].

What is GDScript and why should I use it?

GDScript is Godot's integrated scripting language. It was built from the ground up to maximize Godot's potential in the least amount of code, affording both novice and expert developers alike to capitalize on Godot's strengths as fast as possible. If you've ever written anything in a language like Python before, then you'll feel right at home. For examples and a complete overview of the power GDScript offers you, check out the GDScript scripting guide.

There are several reasons to use GDScript, but the most salient reason is the overall **reduction of complexity**.

The original intent of creating a tightly integrated, custom scripting language for Godot was two-fold: first, it reduces the amount of time necessary to get up and running with Godot, giving developers a rapid way of exposing themselves to the engine with a focus on productivity; second, it reduces the overall burden of maintenance, attenuates the dimensionality of issues, and allows the developers of the engine to focus on squashing bugs and improving features related to the engine core, rather than spending a lot of time trying to get a small set of incremental features working across a large set of languages.

Since Godot is an open source project, it was imperative from the start to prioritize a more integrated and seamless experience over attracting additional users by supporting more familiar programming languages, especially when supporting those more familiar languages would result in a worse experience. We understand if you would rather use another language in Godot (see the list of supported options above). That being said, if you haven't given GDScript a try, try it for **three days**. Just like Godot, once you see how powerful it is and how rapid your development becomes, we think GDScript will grow on you.

More information about getting comfortable with GDScript or dynamically typed languages can be found in the GDScript: An introduction to dynamic languages tutorial.

What were the motivations behind creating GDScript?

In the early days, the engine used the Lua [https://www.lua.org] scripting language. Lua can be fast thanks to LuaJIT, but creating bindings to an object-oriented system (by using fallbacks) was complex and slow and took an enormous amount of code. After some experiments with Python [https://www.python.org], that also proved difficult to embed.

The main reasons for creating a custom scripting language for Godot were:

1. Poor threading support in most script VMs, and Godot uses threads (Lua, Python, Squirrel, JavaScript, ActionScript, etc.).
2. Poor class-extending support in most script VMs, and adapting to the way Godot works is highly inefficient (Lua, Python, JavaScript).
3. Many existing languages have horrible interfaces for binding to C++, resulting in a large amount of code, bugs, bottlenecks, and general inefficiency (Lua, Python, Squirrel, JavaScript, etc.). We wanted to focus on a great engine, not a great number of integrations.
4. No native vector types (Vector3, Transform3D, etc.), resulting in highly reduced performance when using custom types (Lua, Python, Squirrel, JavaScript, ActionScript, etc.).
5. Garbage collector results in stalls or unnecessarily large memory usage (Lua, Python, JavaScript, ActionScript, etc.).
6. Difficulty integrating with the code editor for providing code completion, live editing, etc. (all of them).

GDScript was designed to curtail the issues above, and more.

What 3D model formats does Godot support?

You can find detailed information on supported formats, how to export them from your 3D modeling software, and how to import them for Godot in the Importing 3D scenes documentation.

Will [insert closed SDK such as FMOD, GameWorks, etc.] be supported in Godot?

The aim of Godot is to create a free and open source MIT-licensed engine that is modular and extendable. There are no plans for the core engine development community to support any third-party, closed-source/proprietary SDKs, as integrating with these would go against Godot's ethos.

That said, because Godot is open source and modular, nothing prevents you or anyone else interested in adding those libraries as a module and shipping your game with them, as either open- or closed-source.

To see how support for your SDK of choice could still be provided, look at the Plugins question below.

If you know of a third-party SDK that is not supported by Godot but that offers free and open source integration, consider starting the integration work yourself. Godot is not owned by one person; it belongs to the community, and it grows along with ambitious community contributors like you.

How can I extend Godot?

For extending Godot, like creating Godot Editor plugins or adding support for additional languages, take a look at EditorPlugins and tool scripts.

Also, see the official blog post on GDExtension, a way to develop native extensions for Godot:

- Introducing GDNative's successor, GDExtension [https://godotengine.org/article/introducing-gd-extensions]

You can also take a look at the GDScript implementation, the Godot modules, as well as the Jolt physics engine integration [https://github.com/godot-jolt/godot-jolt] for Godot. This would be a good starting point to see how another third-party library integrates with Godot.

How do I install the Godot editor on my system (for desktop integration)?

Since you don't need to actually install Godot on your system to run it, this means desktop integration is not performed automatically. There are two ways to overcome this. You can install Godot from Steam [https://store.steampowered.com/app/404790/Godot_Engine/] (all platforms), Scoop [https://scoop.sh/] (Windows), Homebrew [https://brew.sh/] (macOS) or Flathub [https://flathub.org/apps/details/org.godotengine.Godot] (Linux). This will automatically perform the required steps for desktop integration.

Alternatively, you can manually perform the steps that an installer would do for you:

Windows

- Move the Godot executable to a stable location (i.e. outside of your Downloads folder), so you don't accidentally move it and break the shortcut in the future.
- Right-click the Godot executable and choose **Create Shortcut**.
- Move the created shortcut to `%APPDATA%\Microsoft\Windows\Start Menu\Programs`. This is the user-wide location for shortcuts that will appear in the Start menu. You can also pin Godot in the task bar by right-clicking the executable and choosing **Pin to Task Bar**.

macOS

Drag the extracted Godot application to `/Applications/Godot.app`, then drag it to the Dock if desired. Spotlight will be able to find Godot as long as it's in `/Applications` or `~/Applications`.

Linux

- Move the Godot binary to a stable location (i.e. outside of your Downloads folder), so you don't accidentally move it and break the shortcut in the future.
- Rename and move the Godot binary to a location present in your `PATH` environment variable. This is typically `/usr/local/bin/godot` or `/usr/bin/godot`. Doing this requires administrator privileges, but this also allows you to run the Godot editor from a terminal by entering `godot`.
  - If you cannot move the Godot editor binary to a protected location, you can keep the binary somewhere in your home directory, and modify the `Path=` line in the `.desktop` file linked below to contain the full *absolute* path to the Godot binary.
- Save this .desktop file [https://raw.githubusercontent.com/godotengine/godot/master/misc/dist/linux/org.godotengine.Godot.desktop] to `$HOME/.local/share/applications/`. If you have administrator privileges, you can also save the `.desktop` file to `/usr/local/share/applications` to make the shortcut available for all users.

Is the Godot editor a portable application?

In its default configuration, Godot is *semi-portable*. Its executable can run from any location (including non-writable locations) and never requires administrator privileges.

However, configuration files will be written to the user-wide configuration or data directory. This is usually a good approach, but this means configuration files will not carry across machines if you copy the folder containing the Godot executable. See File paths in Godot projects for more information.

If *true* portable operation is desired (e.g. for use on a USB stick), follow the steps in Self-contained mode.

Why does Godot prioritize Vulkan and OpenGL over Direct3D?

Godot aims for cross-platform compatibility and open standards first and foremost. OpenGL and Vulkan are the technologies that are both open and available on (nearly) all platforms. Thanks to this design decision, a project developed with Godot on Windows will run out of the box on Linux, macOS, and more.

While Vulkan and OpenGL remain our primary focus for their open standard and cross-platform benefits, Godot 4.3 introduced experimental support for Direct3D 12. This addition aims to enhance performance and compatibility on platforms where Direct3D 12 is prevalent, such as Windows and Xbox. However, Vulkan and OpenGL will continue as the default rendering backends on all platforms, including Windows.

Why does Godot aim to keep its core feature set small?

Godot intentionally does not include features that can be implemented by add-ons unless they are used very often. One example of something not used often is advanced artificial intelligence functionality.

There are several reasons for this:

- **Code maintenance and surface for bugs.** Every time we accept new code in the Godot repository, existing contributors often take the responsibility of maintaining it. Some contributors don't always stick around after getting their code merged, which can make it difficult for us to maintain the code in question. This can lead to poorly maintained features with bugs that are never fixed. On top of that, the "API surface" that needs to be tested and checked for regressions keeps increasing over time.
- **Ease of contribution.** By keeping the codebase small and tidy, it can remain fast and easy to compile from source. This makes it easier for new contributors to get started with Godot, without requiring them to purchase high-end hardware.
- **Keeping the binary size small for the editor.** Not everyone has a fast Internet connection. Ensuring that everyone can download the Godot editor, extract it and run it in less than 5 minutes makes Godot more accessible to developers in all countries.
- **Keeping the binary size small for export templates.** This directly impacts the size of projects exported with Godot. On mobile and web platforms, keeping file sizes low is important to ensure fast installation and loading on underpowered devices. Again, there are many countries where high-speed Internet is not readily available. To add to this, strict data usage caps are often in effect in those countries.

For all the reasons above, we have to be selective of what we can accept as core functionality in Godot. This is why we are aiming to move some core functionality to officially supported add-ons in future versions of Godot. In terms of binary size, this also has the advantage of making you pay only for what you actually use in your project. (In the meantime, you can compile custom export templates with unused features disabled to optimize the distribution size of your project.)

How should assets be created to handle multiple resolutions and aspect ratios?

This question pops up often and it's probably thanks to the misunderstanding created by Apple when they originally doubled the resolution of their devices. It made people think that having the same assets in different resolutions was a good idea, so many continued towards that path. That originally worked to a point and only for Apple devices, but then several Android and Apple devices with different resolutions and aspect ratios were created, with a very wide range of sizes and DPIs.

The most common and proper way to achieve this is to, instead, use a single base resolution for the game and only handle different screen aspect ratios. This is mostly needed for 2D, as in 3D, it's just a matter of camera vertical or horizontal FOV.

1. Choose a single base resolution for your game. Even if there are devices that go up to 1440p and devices that go down to 400p, regular hardware scaling in your device will take care of this at little or no performance cost. The most common choices are either near 1080p (1920x1080) or 720p (1280x720). Keep in mind the higher the resolution, the larger your assets, the more memory they will take and the longer the time it will take for loading.
2. Use the stretch options in Godot; canvas items stretching while keeping aspect ratios works best. Check the Multiple resolutions tutorial on how to achieve this.
3. Determine a minimum resolution and then decide if you want your game to stretch vertically or horizontally for different aspect ratios, or if there is one aspect ratio and you want black bars to appear instead. This is also explained in Multiple resolutions.

4. For user interfaces, use the anchoring to determine where controls should stay and move. If UIs are more complex, consider learning about Containers.

And that's it! Your game should work in multiple resolutions.

When is the next release of Godot out?

When it's ready! See When is the next release out? for more information.

Which Godot version should I use for a new project?

We recommend using Godot 4.x for new projects, but depending on the feature set you need, it may be better to use 3.x instead. See Which version should I use for a new project? for more information.

Should I upgrade my project to use new Godot versions?

Some new versions are safer to upgrade to than others. In general, whether you should upgrade depends on your project's circumstances. See Should I upgrade my project to use new engine versions? for more information.

I would like to contribute! How can I get started?

Awesome! As an open source project, Godot thrives off of the innovation and the ambition of developers like you.

The best way to start contributing to Godot is by using it and reporting any issues [https://github.com/godotengine/godot/issues] that you might experience. A good bug report with clear reproduction steps helps your fellow contributors fix bugs quickly and efficiently. You can also report issues you find in the online documentation [https://github.com/godotengine/godot-docs/issues].

If you feel ready to submit your first PR, pick any issue that resonates with you from one of the links above and try your hand at fixing it. You will need to learn how to compile the engine from sources, or how to build the documentation. You also need to get familiar with Git, a version control system that Godot developers use.

We explain how to work with the engine source, how to edit the documentation, and what other ways to contribute are there in our documentation for contributors.

I have a great idea for Godot. How can I share it?

We are always looking for suggestions about how to improve the engine. User feedback is the main driving force behind our decision-making process, and limitations that you might face while working on your project are a great data point for us when considering engine enhancements.

If you experience a usability problem or are missing a feature in the current version of Godot, start by discussing it with our community [https://godotengine.org/community/]. There may be other, perhaps better, ways to achieve the desired result that community members could suggest. And you can learn if other users experience the same issue, and figure out a good solution together.

If you come up with a well-defined idea for the engine, feel free to open a proposal issue [https://github.com/godotengine/godot-proposals/issues]. Try to be specific and concrete while describing your problem and your proposed solution — only actionable proposals can be considered. It is not required, but if you want to implement it yourself, that's always appreciated!

If you only have a general idea without specific details, you can open a proposal discussion [https://github.com/godotengine/godot-proposals/discussions]. These can be anything you want, and allow for a free-form discussion in search of a solution. Once you find one, a proposal issue can be opened.

Please, read the readme [https://github.com/godotengine/godot-proposals/blob/master/README.md] document before creating a proposal to learn more about the process.

Is it possible to use Godot to create non-game applications?

Yes! Godot features an extensive built-in UI system, and its small distribution size can make it a suitable alternative to frameworks like Electron or Qt.

When creating a non-game application, make sure to enable low-processor mode in the Project Settings to decrease CPU and GPU usage.

Check out Material Maker [https://github.com/RodZill4/material-maker] and Pixelorama [https://github.com/Orama-Interactive/Pixelorama] for examples of open source applications made with Godot.

Is it possible to use Godot as a library?

Godot is meant to be used with its editor. We recommend you give it a try, as it will most likely save you time in the long term. There are no plans to make Godot usable as a library, as it would make the rest of the engine more convoluted and difficult to use for casual users.

If you want to use a rendering library, look into using an established rendering engine instead. Keep in mind rendering engines usually have smaller communities compared to Godot. This will make it more difficult to find answers to your questions.

What user interface toolkit does Godot use?

Godot does not use a standard GUI toolkit like GTK, Qt or wxWidgets. Instead, Godot uses its own user interface toolkit, rendered using OpenGL ES or Vulkan. This toolkit is exposed in the form of Control nodes, which are used to render the editor (which is written in C++). These Control nodes can also be used in projects from any scripting language supported by Godot.

This custom toolkit makes it possible to benefit from hardware acceleration and have a consistent appearance across all platforms. On top of that, it doesn't have to deal with the LGPL licensing caveats that come with GTK or Qt. Lastly, this means Godot is "eating its own dog food" since the editor itself is one of the most complex users of Godot's UI system.

This custom UI toolkit can't be used as a library, but you can still use Godot to create non-game applications by using the editor.

Why does Godot use the SCons build system?

Godot uses the SCons [https://www.scons.org/] build system. There are no plans to switch to a different build system in the near future. There are many reasons why we have chosen SCons over other alternatives. For example:

- Godot can be compiled for a dozen different platforms: all PC platforms, all mobile platforms, many consoles, and WebAssembly.
- Developers often need to compile for several of the platforms **at the same time**, or even different targets of the same platform. They can't afford reconfiguring and rebuilding the project each time. SCons can do this with no sweat, without breaking the builds.
- SCons will *never* break a build no matter how many changes, configurations, additions, removals etc.
- Godot's build process is not simple. Several files are generated by code (binders), others are parsed (shaders), and others need to offer customization (modules). This requires complex logic which is easier to write in an actual programming language (like Python) rather than using a mostly macro-based language only meant for building.
- Godot's build process makes heavy use of cross-compiling tools. Each platform has a specific detection process, and all these must be handled as specific cases with special code written for each.

Please try to keep an open mind and get at least a little familiar with SCons if you are planning to build Godot yourself.

Why does Godot not use STL (Standard Template Library)?

Like many other libraries (Qt as an example), Godot does not make use of STL (with a few exceptions such as threading primitives). We believe STL is a great general-purpose library, but we had special requirements for Godot.

- STL templates create very large symbols, which results in huge debug binaries. We use few templates with very short names instead.
- Most of our containers cater to special needs, like Vector, which uses copy on write and we use to pass data around, or the RID system, which requires O(1) access time for performance. Likewise, our hash map implementations are designed to integrate seamlessly with internal engine types.
- Our containers have memory tracking built-in, which helps better track memory usage.
- For large arrays, we use pooled memory, which can be mapped to either a preallocated buffer or virtual memory.
- We use our custom String type, as the one provided by STL is too basic and lacks proper internationalization support.

Why does Godot not use exceptions?

We believe games should not crash, no matter what. If an unexpected situation happens, Godot will print an error (which can be traced even to script), but then it will try to recover as gracefully as possible and keep going.

Additionally, exceptions significantly increase the binary size for the executable and result in increased compile times.

Does Godot use an ECS (Entity Component System)?

Godot does **not** use an ECS and relies on inheritance instead. While there is no universally better approach, we found that using an inheritance-based approach resulted in better usability while still being fast enough for most use cases.

That said, nothing prevents you from making use of composition in your project by creating child Nodes with individual scripts. These nodes can then be added and removed at run-time to dynamically add and remove behaviors.

More information about Godot's design choices can be found in this article [https://godotengine.org/article/why-isnt-godot-ecs-based-game-engine].

Why does Godot not force users to implement DOD (Data-Oriented Design)?

While Godot internally attempts to use cache coherency as much as possible, we believe users don't need to be forced to use DOD practices.

DOD is mostly a cache coherency optimization that can only provide significant performance improvements when dealing with dozens of thousands of objects which are processed every frame with little modification. That is, if you are moving a few hundred sprites or enemies per frame, DOD won't result in a meaningful improvement in performance. In such a case, you should consider a different approach to optimization.

The vast majority of games do not need this and Godot provides handy helpers to do the job for most cases when you do.

If a game needs to process such a large amount of objects, our recommendation is to use C++ and GDExtensions for performance-heavy tasks and GDScript (or C#) for the rest of the game.

How can I support Godot development or contribute?

See How to contribute.

Who is working on Godot? How can I contact you?

See the corresponding page on the Godot website [https://godotengine.org/contact].

Complying with licenses

Warning

The recommendations in this page **are not legal advice.** They are provided in good faith to help users navigate license attribution requirements.

What are licenses?

Godot is created and distributed under the MIT License [https://opensource.org/licenses/MIT]. It doesn't have a sole owner, as every contributor that submits code to the project does it under this same license and keeps ownership of their contribution.

The license is the legal requirement for you (or your company) to use and distribute the software (and derivative projects, including games made with it). Your game or project can have a different license, but it still needs to comply with the original one.

Note

This section covers compliance with licenses from a user perspective. If you are interested in licence compliance as a contributor, you can find guidelines here.

Tip

Alongside the Godot license text, remember to also list third-party notices for assets you're using, such as textures, models, sounds, music and fonts. This includes free assets, which often come with licenses that require attribution.

Requirements

In the case of the MIT license, the only requirement is to include the license text somewhere in your game or derivative project.

This text reads as follows:

```
This game uses Godot Engine, available under the following
license:

Copyright (c) 2014-present Godot Engine contributors.
Copyright (c) 2007-2014 Juan Linietsky, Ariel Manzur.

Permission is hereby granted, free of charge, to any person
obtaining a copy
of this software and associated documentation files (the
"Software"), to deal
in the Software without restriction, including without
limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell
copies of the Software, and to permit persons to whom the
Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be
included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES
```

```
OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE
SOFTWARE.
```

Beside its own MIT license, Godot includes code from a number of third-party libraries. See Third-party licenses for details.

Note

Your games do not need to be under the same license. You are free to release your Godot projects under any license and to create commercial games with the engine.

Inclusion

The license text must be made available to the user. The license doesn't specify how the text has to be included, but here are the most common approaches (you only need to implement one of them, not all).

Credits screen

Include the above license text somewhere in the credits screen. It can be at the bottom after showing the rest of the credits. Most large studios use this approach with open source licenses.

Licenses screen

Some games have a special menu (often in the settings) to display licenses. This menu is typically accessed with a button called **Third-**

party Licenses or **Open Source Licenses**.

Output log

Printing the license text using the print() function may be enough on platforms where a global output log is readable. This is the case on desktop platforms, Android and HTML5 (but not iOS).

Accompanying file

If the game is distributed on desktop platforms, a file containing the license text can be added to the software that is installed to the user PC.

Printed manual

If the game includes a printed manual, the license text can be included there.

Link to the license

The Godot Engine developers consider that a link to `godotengine.org/license` in your game documentation or credits would be an acceptable way to satisfy the license terms.

Tip

Godot provides several methods to get license information in the Engine singleton. This allows you to source the license information directly from the engine binary, which prevents the information from becoming outdated if you update engine versions.

For the engine itself:

- Engine.get_license_text

For third-party components used by the engine:

- Engine.get_license_info
- Engine.get_copyright_info

Third-party licenses

Godot itself contains software written by third parties [https://github.com/godotengine/godot/blob/master/thirdparty/README.md], which is compatible with, but not covered by Godot's MIT license.

Many of these dependencies are distributed under permissive open source licenses which require attribution by explicitly citing their copyright statement and license text in the final product's documentation.

Given the scope of the Godot project, this is fairly difficult to do thoroughly. For the Godot editor, the full documentation of third-party copyrights and licenses is provided in the COPYRIGHT.txt [https://github.com/godotengine/godot/blob/master/COPYRIGHT.txt] file.

A good option for end users to document third-party licenses is to include this file in your project's distribution, which you can e.g. rename to `GODOT_COPYRIGHT.txt` to prevent any confusion with your own code and assets.

Godot release policy

Godot's release policy is in constant evolution. The description below provides a general idea of what to expect, but what will actually happen depends on the choices of core contributors and the needs of the community at a given time.

Godot versioning

Godot loosely follows Semantic Versioning [https://semver.org/] with a `major.minor.patch` versioning system, albeit with an interpretation of each term adapted to the complexity of a game engine:

- The `major` version is incremented when major compatibility breakages happen which imply significant porting work to move projects from one major version to another.

  For example, porting Godot projects from Godot 3.x to Godot 4.x requires running the project through a conversion tool, and then performing a number of further adjustments manually for what the tool could not do automatically.

- The `minor` version is incremented for feature releases that do not break compatibility in a major way. Minor compatibility breakage in very specific areas *may* happen in minor versions, but the vast majority of projects should not be affected or require significant porting work.

  This is because Godot, as a game engine, covers many areas like rendering, physics, and scripting. Fixing bugs or implementing new features in one area might sometimes require changing a feature's behavior or modifying a class's interface, even if the rest of the engine API remains backwards compatible.

Tip

Upgrading to a new minor version is recommended for all users, but some testing is necessary to ensure that your project still behaves as expected.

- The `patch` version is incremented for maintenance releases which focus on fixing bugs and security issues, implementing new requirements for platform support, and backporting safe usability enhancements. Patch releases are backwards compatible.

  Patch versions may include minor new features which do not impact the existing API, and thus have no risk of impacting existing projects.

Tip

Updating to new patch versions is therefore considered safe and strongly recommended to all users of a given stable branch.

We call `major.minor` combinations *stable branches*. Each stable branch starts with a `major.minor` release (without the `0` for `patch`) and is further developed for maintenance releases in a Git branch of the same name (for example patch updates for the 4.0 stable branch are developed in the `4.0` Git branch).

Release support timeline

Stable branches are supported *at least* until the next stable branch is released and has received its first patch update. In practice, we support stable branches on a *best effort* basis for as long as they have active users who need maintenance updates.

Whenever a new major version is released, we make the previous stable branch a long-term supported release, and do our best to provide fixes for issues encountered by users of that branch who cannot port complex projects to the new major version. This was the case for the 2.1 branch, and is the case for the 3.6 branch.

In a given minor release series, only the latest patch release receives support. If you experience an issue using an older patch release, please upgrade to the latest patch release of that series and test again before reporting an issue on GitHub.

Version	Release date	Support level
Godot 4.3 (*master*)	June 2024 (estimate)	● *Development.* Receives new features, usability and performance improvements, as well as bug fixes, while under development.
Godot 4.2	November 2023	● Receives fixes for bugs and security issues, as well as patches that enable platform support.
Godot 4.1	July 2023	● Receives fixes for bugs and security issues, as well as patches that enable platform support.

Godot 4.0	March 2023	🔴 No longer supported (last update: 4.0.4).
Godot 3.6 (*3.x*, LTS)	Q1 2024 (estimate)	🟢 *Beta.* Receives new features, usability and performance improvements, as well as bug fixes, while under development.
Godot 3.5	August 2022	🟢 Receives fixes for bugs and security issues, as well as patches that enable platform support.
Godot 3.4	November 2021	🔴 No longer supported (last update: 3.4.5).
Godot 3.3	April 2021	🔴 No longer supported (last update: 3.3.4).
Godot 3.2	January 2020	🔴 No longer supported (last update: 3.2.3).
Godot 3.1	March 2019	🔴 No longer supported (last update: 3.1.2).
Godot 3.0	January 2018	🔴 No longer supported (last update: 3.0.6).
Godot 2.1	July 2016	🔴 No longer supported (last update: 2.1.6).

Godot 2.0	February 2016	● No longer supported (last update: 2.0.4.1).
Godot 1.1	May 2015	● No longer supported.
Godot 1.0	December 2014	● No longer supported.

Legend: ● Full support – ● Partial support – ● No support (end of life) – ● Development version

Pre-release Godot versions aren't intended to be used in production and are provided for testing purposes only.

See also

See Upgrading from Godot 3 to Godot 4 for instructions on migrating a project from Godot 3.x to 4.x.

Which version should I use for a new project?

We recommend using Godot 4.x for new projects, as the Godot 4.x series will be supported long after 3.x stops receiving updates in the future. One caveat is that a lot of third-party documentation hasn't been updated for Godot 4.x yet. If you have to follow a tutorial designed for Godot 3.x, we recommend keeping Upgrading from Godot 3 to Godot 4 open in a separate tab to check which methods have been renamed (if you get a script error while trying to use a specific node or method that was renamed in Godot 4.x).

If your project requires a feature that is missing in 4.x (such as GLES2/WebGL 1.0), you should use Godot 3.x for a new project instead.

Should I upgrade my project to use new engine versions?

Note

Upgrading software while working on a project is inherently risky, so consider whether it's a good idea for your project before attempting an upgrade. Also, make backups of your project or use version control to prevent losing data in case the upgrade goes wrong.

That said, we do our best to keep minor and especially patch releases compatible with existing projects.

The general recommendation is to upgrade your project to follow new *patch* releases, such as upgrading from 4.0.2 to 4.0.3. This ensures you get bug fixes, security updates and platform support updates (which is especially important for mobile platforms). You also get continued support, as only the last patch release receives support on official community platforms.

For *minor* releases, you should determine whether it's a good idea to upgrade on a case-by-case basis. We've made a lot of effort in making the upgrade process as seamless as possible, but some breaking changes may be present in minor releases, along with a greater risk of regressions. Some fixes included in minor releases may also change a class' expected behavior as required to fix some bugs. This is especially the case in classes marked as *experimental* in the documentation.

Major releases bring a lot of new functionality, but they also remove previously existing functionality and may raise hardware requirements. They also require much more work to upgrade to compared to minor releases. As a result, we recommend sticking with the major release you've started your project with if you are happy with how your project currently works. For example, if your project was started with 3.5, we recommend upgrading to 3.5.2 and possibly 3.6 in the future, but not to 4.0+, unless your project really needs the new features that come with 4.0+.

When is the next release out?

While Godot contributors aren't working under any deadlines, we strive to publish minor releases relatively frequently.

In particular, after the very length release cycle for 4.0, we are pivoting to a faster paced development workflow, 4.1 released 4 months after 4.0, and 4.2 released 4 months after 4.1

Frequent minor releases will enable us to ship new features faster (possibly as experimental), get user feedback quickly, and iterate to improve those features and their usability. Likewise, the general user experience will be improved more steadily with a faster path to the end users.

Maintenance (patch) releases are released as needed with potentially very short development cycles, to provide users of the current stable branch with the latest bug fixes for their production needs.

The 3.6 release is still planned and should be the last stable branch of Godot 3.x. It will be a Long-Term Support (LTS) release, which we plan to support for as long as users still need it (due to missing features in Godot 4.x, or having published games which they need to keep updating for platform requirements).

What are the criteria for compatibility across engine versions?

Note

This section is intended to be used by contributors to determine which changes are safe for a given release. The list is not exhaustive; it only outlines the most common situations encountered during Godot's development.

The following changes are acceptable in patch releases:

- Fixing a bug in a way that has no major negative impact on most projects, such as a visual or physics bug. Godot's physics engine is not deterministic, so physics bug fixes are not considered to break compatibility. If fixing a bug has a negative impact that could impact a lot of projects, it should be made optional (e.g. using a project setting or separate method).
- Adding a new optional parameter to a method.
- Small-scale editor usability tweaks.

Note that we tend to be more conservative with the fixes we allow in each subsequent patch release. For instance, 4.0.1 may receive more impactful fixes than 4.0.4 would.

The following changes are acceptable in minor releases, but not patch releases:

- Significant new features.
- Renaming a method parameter. In C#, method parameters can be passed by name (but not in GDScript). As a result, this can break some projects that use C#.
- Deprecating a method, member variable, or class. This is done by adding a deprecated flag to its class reference, which will show up in the editor. When a method is marked as deprecated, it's slated to be removed in the next *major* release.
- Changes that affect the default project theme's visuals.
- Bug fixes which significantly change the behavior or the output, with the aim to meet user expectations better. In comparison, in patch releases, we may favor keeping a buggy behavior so we don't break existing projects which likely already rely on the bug or use a workaround.
- Performance optimizations that result in visual changes.

The following changes are considered **compatibility-breaking** and can only be performed in a new major release:

- Renaming or removing a method, member variable, or class.
- Modifying a node's inheritance tree by making it inherit from a different class.
- Changing the default value of a project setting value in a way that affects existing projects. To only affect new projects, the project manager should write a modified `project.godot` instead.

Since Godot 5.0 hasn't been branched off yet, we currently discourage making compatibility-breaking changes of this kind.

Note

When modifying a method's signature in any fashion (including adding an optional parameter), a GDExtension compatibility method must be created. This ensures that existing GDExtensions continue to work across patch and minor releases, so that users don't have to recompile them. See Handling compatibility breakages for more information.

Documentation changelog

The documentation is continually being improved. New releases include new pages, fixes and updates to existing pages, and many updates to the class reference. Below is a list of new pages added since version 3.0.

Note

This document only contains new pages so not all changes are reflected, many pages have been substantially updated but are not reflected in this document.

New pages since version 4.1

C#

- C# diagnostics

Development

- 2D coordinate systems and 2D transforms

Migrating

- Upgrading from Godot 4.1 to Godot 4.2

I/O

- Runtime file loading and saving

Platform-specific

- Godot Android library

New pages since version 4.0

Development

- Internal rendering architecture
- Using sanitizers

Migrating

- Upgrading from Godot 4.0 to Godot 4.1

Physics

- Troubleshooting physics issues

New pages since version 3.6

2D

- 2D antialiasing

3D

- 3D antialiasing
- Faking global illumination
- Introduction to global illumination
- Mesh level of detail (LOD)

- Occlusion culling
- Signed distance field global illumination (SDFGI)
- Using decals
- Visibility ranges (HLOD)
- Volumetric fog and fog volumes
- Variable rate shading
- Physical light and camera units

Animation

- Creating movies

Assets pipeline

- Retargeting 3D Skeletons

Development

- Custom platform ports

Migrating

- Upgrading from Godot 3 to Godot 4

Physics

- Large world coordinates

Scripting

- Custom performance monitors
- C# collections
- C# global classes

- C# Variant

Shaders

- Using compute shaders

Workflow

- Pull request review process

XR

- Introducing XR tools
- The XR action map
- Deploying to Android

New pages since version 3.5

None.

New pages since version 3.4

3D

- 3D text

Animation

- Playing videos

Editor

- Managing editor features

New pages since version 3.3

C++

- C++ usage guidelines

GDScript

- GDScript documentation comments

New pages since version 3.2

3D

- 3D rendering limitations

About

- Troubleshooting
- List of features
- Godot release policy

Best practices

- Version control systems

Community

- Best practices for engine contributors

- Bisecting regressions
- Editor and documentation localization

Development

- Introduction to editor development
- Editor style guide
- Common engine methods and macros
- Validation layers
- GDScript grammar
- Configuring an IDE: Code::Blocks

Editor

- Default editor shortcuts
- Using the Web editor

Export

- Exporting for dedicated servers

Input

- Controllers, gamepads, and joysticks

Math

- Random number generation

Platform-specific

- Plugins for iOS
- Creating iOS plugins

- HTML5 shell class reference

Physics

- Collision shapes (2D)
- Collision shapes (3D)

Shaders

- Shaders style guide

Scripting

- Debugger panel
- Creating script templates
- Evaluating expressions
- What is GDExtension?
- GDScript warning system (split from Static typing in GDScript)

User Interface (UI)

- Control node gallery

New pages since version 3.1

Project workflow

- Gradle builds for Android

2D

- 2D sprite animation

Audio

- Recording with microphone
- Sync the gameplay with audio and music

Math

- Beziers, curves and paths
- Interpolation

Inputs

- Input examples

Internationalization

- Localization using gettext

Shading

- Your First Shader Series:
    - Introduction to shaders
    - Your first 2D shader
    - Your first 3D shader
    - Your second 3D shader
- Using VisualShaders

Networking

- WebRTC

Plugins

- Godot Android plugins
- Inspector plugins
- Visual Shader plugins

Multi-threading

- Using multiple threads

Creating content

Procedural geometry series:

  - Procedural geometry
  - Using the ArrayMesh
  - Using the SurfaceTool
  - Using the MeshDataTool
  - Using ImmediateMesh

Optimization

- Optimization using MultiMeshes
- Optimization using Servers

Legal

- Complying with licenses

New pages since version 3.0

Step by step

- Using signals
- Exporting

Scripting

- Static typing in GDScript

Project workflow

Best Practices:

- Introduction
- Applying object-oriented principles in Godot
- Scene organization
- When to use scenes versus scripts
- Autoloads versus regular nodes
- When and how to avoid using nodes for everything
- Godot interfaces
- Godot notifications
- Data preferences
- Logic preferences

2D

- 2D lights and shadows
- 2D meshes

3D

- Prototyping levels with CSG
- Animating thousands of fish with MultiMeshInstance3D
- Controlling thousands of fish with Particles

Physics

- Ragdoll system
- Using SoftBody

Animation

- 2D skeletons
- Using AnimationTree

GUI

- Using Containers

Viewports

- Using a SubViewport as a texture
- Custom post-processing

Shading

- Converting GLSL to Godot shaders
- Advanced post-processing

Shading Reference:

- Introduction to shaders
- Shading language
- Spatial shaders
- CanvasItem shaders
- Particle shaders

Plugins

- Making main screen plugins
- 3D gizmo plugins

Platform-specific

- Custom HTML page for Web export

Multi-threading

- Thread-safe APIs

Creating content

- Making trees

Miscellaneous

- Fixing jitter, stutter and input lag
- Running code in the editor
- Change scenes manually

Compiling

- Optimizing a build for size
- Compiling with PCK encryption key

Engine development

- Binding to external libraries

Introduction

This series will introduce you to Godot and give you an overview of its features.

In the following pages, you will get answers to questions such as "Is Godot for me?" or "What can I do with Godot?". We will then introduce the engine's most essential concepts, run you through the editor's interface, and give you tips to make the most of your time learning it.

- Introduction to Godot
- Learn to code with GDScript
- Overview of Godot's key concepts
- First look at Godot's interface
- Learning new features
- Godot's design philosophy

Introduction to Godot

This article is here to help you figure out whether Godot might be a good fit for you. We will introduce some broad features of the engine to give you a feel for what you can achieve with it and answer questions such as "what do I need to know to get started?".

This is by no means an exhaustive overview. We will introduce many more features in this getting started series.

What is Godot?

Godot is a general-purpose 2D and 3D game engine designed to support all sorts of projects. You can use it to create games or applications you can then release on desktop or mobile, as well as on the web.

You can also create console games with it, although you either need strong programming skills or a developer to port the game for you.

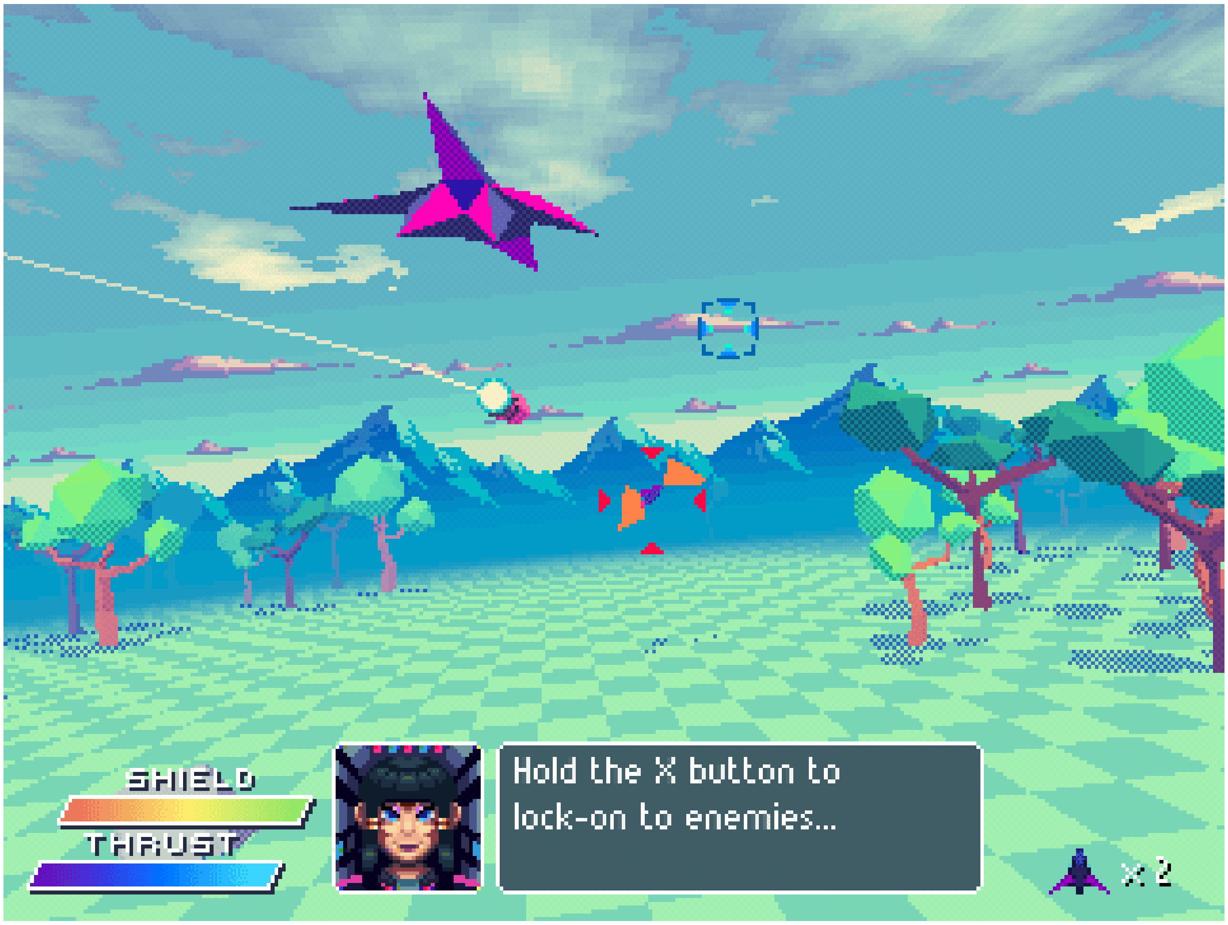
Note

The Godot team can't provide an open source console export due to the licensing terms imposed by console manufacturers. Regardless of the engine you use, though, releasing games on consoles is always a lot of work. You can read more on that here: Console support in Godot.

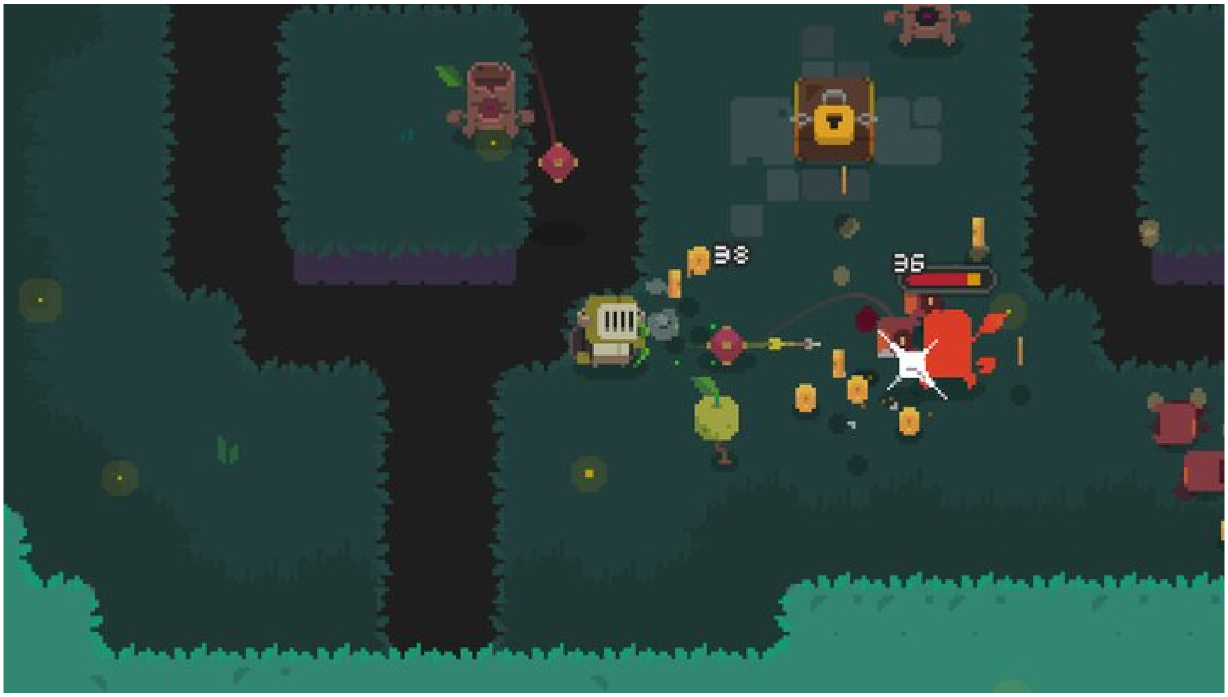
What can the engine do?

Godot was initially developed in-house by an Argentinian game studio. Its development started in 2001, and the engine was rewritten and improved tremendously since its open source release in 2014.

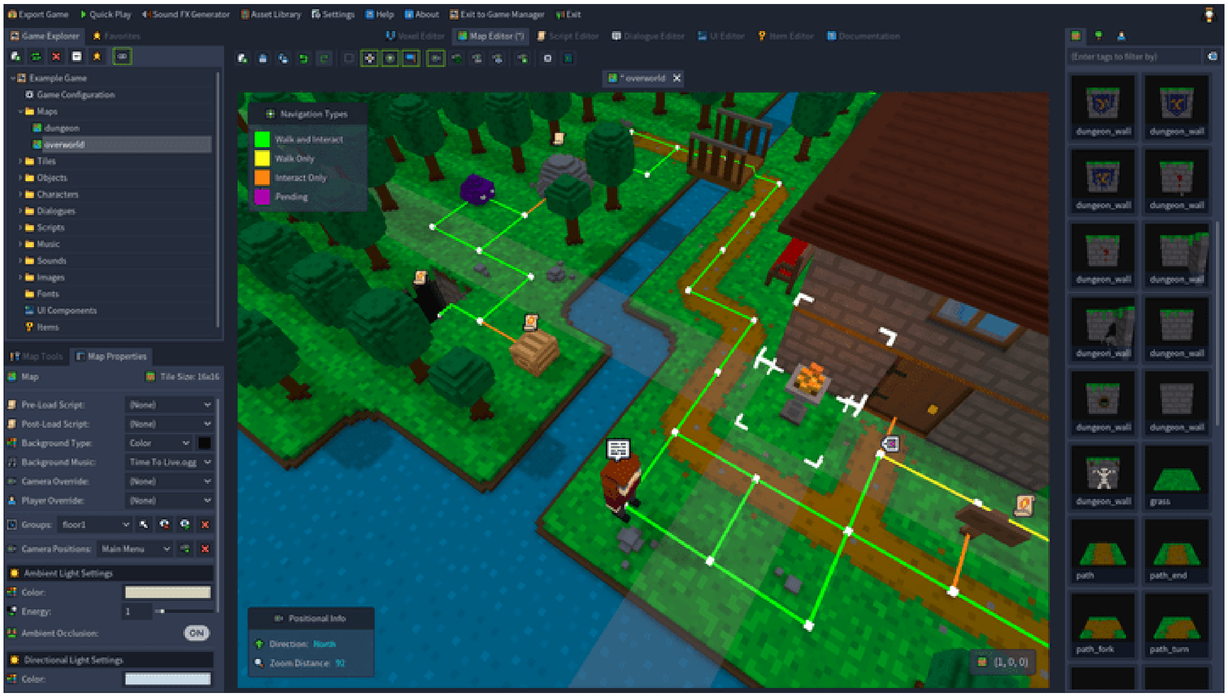
Some examples of games created with Godot include Ex-Zodiac and Helms of Fury.


SHIELD
THRUST
x 2

Hold the X button to lock-on to enemies...



As for applications, the open source pixel art drawing program Pixelorama is powered by Godot, and so is the voxel RPG creator RPG in a box.

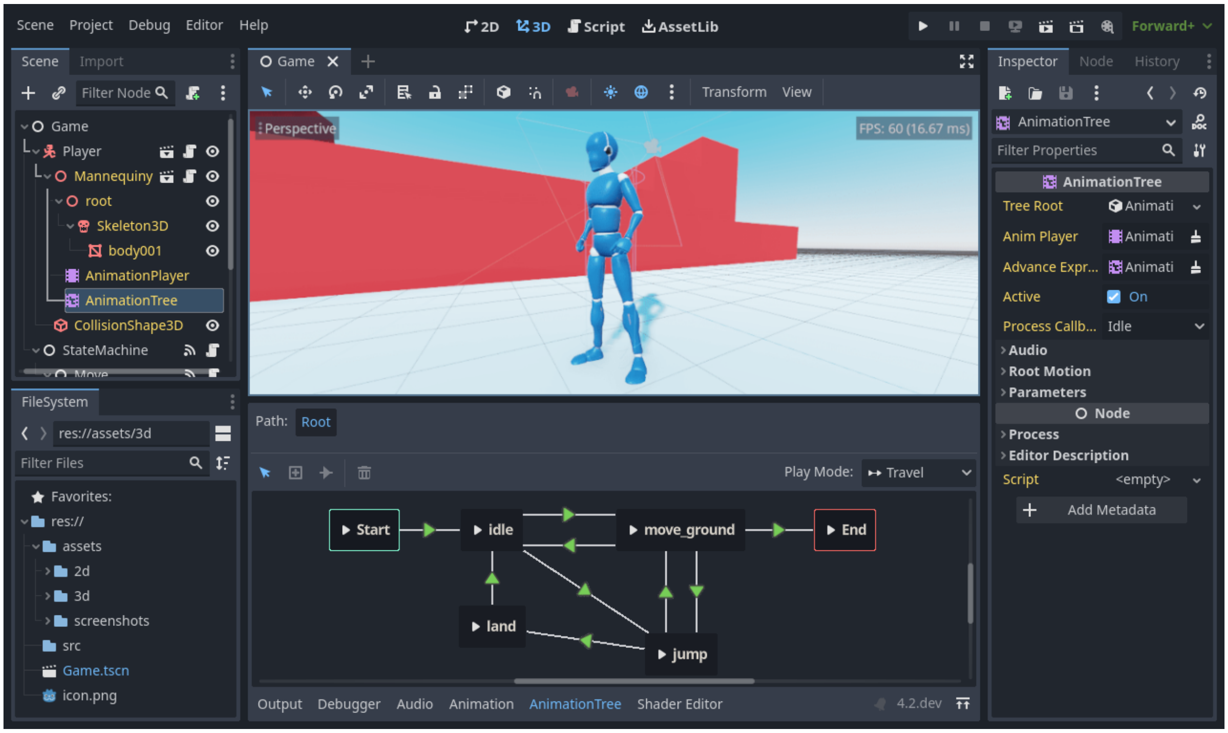

Export Game
Quick Play
Sound FX Generator
Asset Library
Settings
Help
About
Exit to Game Manager
Exit
Game Explorer
Favorites
Voxel Editor
Map Editor (*)
Script Editor
Dialogue Editor
UI Editor
Item Editor
Documentation
Example Game
Game Configuration
Maps
dungeon
overworld
Tiles
Objects
Characters
Dialogues
Scripts
Music
Sounds
Images
Fonts
UI Components
Items
* overworld
Navigation Types
Walk and Interact
Walk Only
Interact Only
Pending
Positional Info
(1, 0, 0)
[Enter tags to filter by]
dungeon_wall
dungeon_wall
dungeon_wall
dungeon_wall
dungeon_wall
dungeon_wall
dungeon_wall
dungeon_wall
dungeon_wall
dungeon_wall
dungeon_wall
grass
path
path_end
path_fork
path_turn
Map Tools
Map Properties
Map
Ambient Light Settings
Directional Light Settings

Tile Size: 16x16
Pre-Load Script: (None)
Post-Load Script: (None)
Background Type: Color
Background Music: Time To Live.ogg
Camera Override: (None)
Player Override: (None)
Groups: floor1
Camera Positions: Main Menu
Color:
Energy: 1
Ambient Occlusion: ON
Color:
Direction: North
Zoom Distance: 92

You can find many more examples in the official showcase videos [https://www.youtube.com/playlist?list=PLeG_dAglpVo6EpaO9A1nkwJZOwrfiLdQ8].

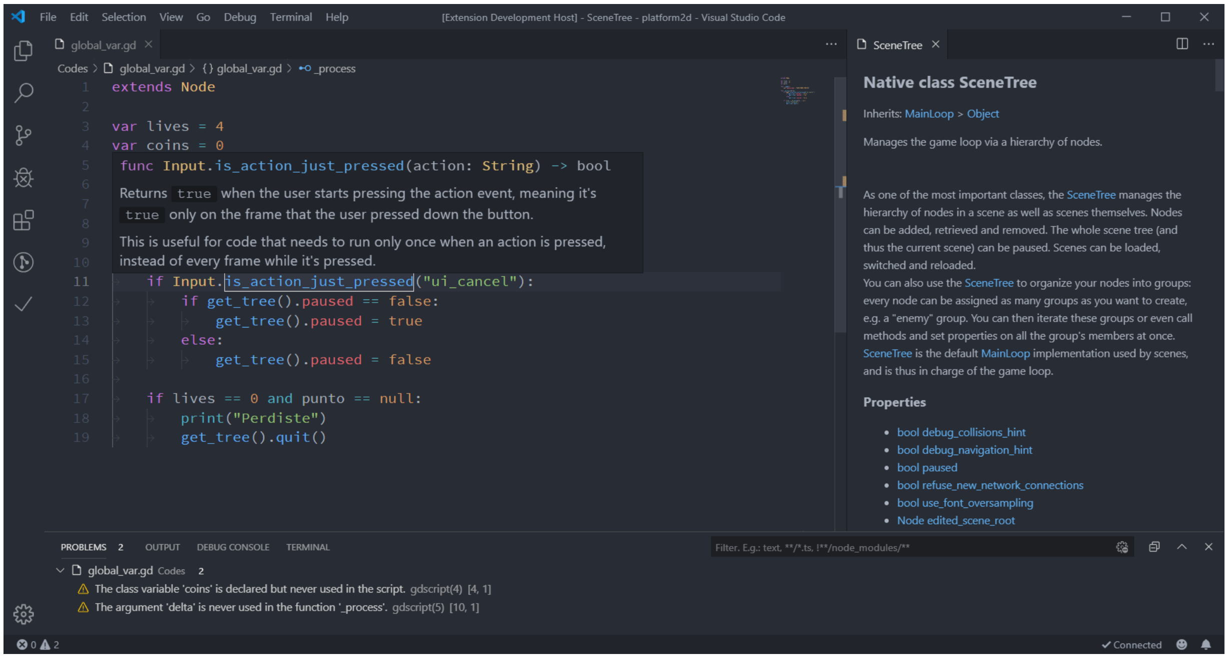
How does it work and look?

Godot comes with a fully-fledged game editor with integrated tools to answer the most common needs. It includes a code editor, an animation editor, a tilemap editor, a shader editor, a debugger, a profiler, and more.


Scene Project Debug Editor Help
2D 3D Script AssetLib
Forward+
Scene Import
Game
Player
Mannequiny
root
Skeleton3D
body001
AnimationPlayer
AnimationTree
CollisionShape3D
StateMachine
FileSystem
res://assets/3d
Filter Files
Favorites:
res://
assets
2d
3d
screenshots
src
Game.tscn
icon.png
Game
Transform View
Perspective
FPS: 60 (16.67 ms)
Path: Root
Play Mode: Travel
Start
idle
move_ground
End
land
jump
Output Debugger Audio Animation AnimationTree Shader Editor
4.2.dev
Inspector Node History
AnimationTree
Filter Properties
Tree Root Animati
Anim Player Animati
Advance Expr... Animati
Active On
Process Callb... Idle
Audio
Root Motion
Parameters
Node
Process
Editor Description
Script <empty>
Add Metadata

The team strives to offer a feature-rich game editor with a consistent user experience. While there is always room for improvement, the user interface keeps getting refined.

Of course, if you prefer, you can work with external programs. We officially support importing 3D scenes designed in Blender [https://www.blender.org/] and maintain plugins to code in VSCode [https://github.com/godotengine/godot-vscode-plugin] and Emacs [https://github.com/godotengine/emacs-gdscript-mode] for GDScript and C#. We also support Visual Studio for C# on Windows.


File
Edit
Selection
View
Go
Debug
Terminal
Help
[Extension Development Host] - SceneTree - platform2d - Visual Studio Code
global_var.gd
Codes > global_var.gd > {} global_var.gd > _process
SceneTree
PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
Filter. E.g.: text, **/*.ts, !**/node_modules/**
0 2
Connected

```
extends Node

var lives = 4
var coins = 0
```

```
func Input.is_action_just_pressed(action: String) -> bool
```

Returns `true` when the user starts pressing the action event, meaning it's `true` only on the frame that the user pressed down the button.

This is useful for code that needs to run only once when an action is pressed, instead of every frame while it's pressed.

```
	if Input.is_action_just_pressed("ui_cancel"):
		if get_tree().paused == false:
			get_tree().paused = true
		else:
			get_tree().paused = false

	if lives == 0 and punto == null:
		print("Perdiste")
		get_tree().quit()
```

Native class SceneTree

Inherits: MainLoop > Object

Manages the game loop via a hierarchy of nodes.

As one of the most important classes, the SceneTree manages the hierarchy of nodes in a scene as well as scenes themselves. Nodes can be added, retrieved and removed. The whole scene tree (and thus the current scene) can be paused. Scenes can be loaded, switched and reloaded.
You can also use the SceneTree to organize your nodes into groups: every node can be assigned as many groups as you want to create, e.g. a "enemy" group. You can then iterate these groups or even call methods and set properties on all the group's members at once.
SceneTree is the default MainLoop implementation used by scenes, and is thus in charge of the game loop.

Properties

- bool debug_collisions_hint
- bool debug_navigation_hint
- bool paused
- bool refuse_new_network_connections
- bool use_font_oversampling
- Node edited_scene_root

```
global_var.gd Codes 2
⚠ The class variable 'coins' is declared but never used in the script. gdscript(4) [4, 1]
⚠ The argument 'delta' is never used in the function '_process'. gdscript(5) [10, 1]
```

Programming languages

Let's talk about the available programming languages.

You can code your games using GDScript, a Godot-specific and tightly integrated language with a lightweight syntax, or C#, which is popular in the games industry. These are the two main scripting languages we support.

With the GDExtension technology, you can also write gameplay or high-performance algorithms in C or C++ without recompiling the engine. You can use this technology to integrate third-party libraries and other Software Development Kits (SDK) in the engine.

Of course, you can also directly add modules and features to the engine, as it's completely free and open source.

What do I need to know to use Godot?

Godot is a feature-packed game engine. With its thousands of features, there is a lot to learn. To make the most of it, you need good programming foundations. While we try to make the engine accessible, you will benefit a lot from knowing how to think like a programmer first.

Godot relies on the object-oriented programming paradigm. Being comfortable with concepts such as classes and objects will help you code efficiently in it.

If you are entirely new to programming, we recommend following the CS50 open courseware [https://cs50.harvard.edu/x] from Harvard University. It's a great free course that will teach you everything you need to know to be off to a good start. It will save you countless hours and hurdles learning any game engine afterward.

Note

In CS50, you will learn multiple programming languages. Don't be afraid of that: programming languages have many similarities. The skills you learn with one language transfer well to others.

We will provide you with more Godot-specific learning resources in Learning new features.

In the next part, you will get an overview of the engine's essential concepts.

Learn to code with GDScript

In Godot, you can write code using the GDScript and C# programming languages.

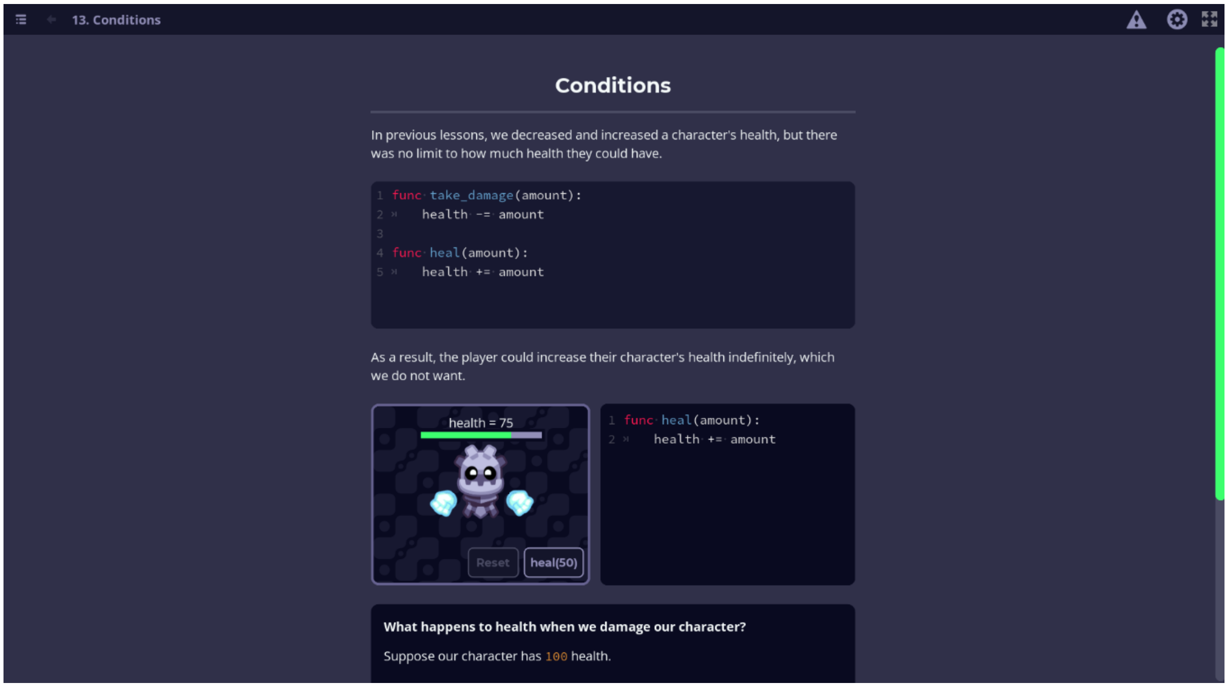
If you are new to programming, we recommend starting with GDScript because we designed it to be simpler than all-purpose languages like C#. It will be both faster and easier to learn.

While GDScript is a language specific to Godot, the techniques you will learn with it will apply to other programming languages.

Note that it is completely normal for a programmer to learn and use multiple languages. Programming languages have more similarities than differences, so once you know one, you can learn another much faster.

Learn in your browser with the GDScript app

To learn GDScript, you can use the app Learn GDScript From Zero. It is a complete beginner course with interactive practices you can do right in your browser.



Conditions

In previous lessons, we decreased and increased a character's health, but there was no limit to how much health they could have.

```
func take_damage(amount):
    health -= amount

func heal(amount):
    health += amount
```

As a result, the player could increase their character's health indefinitely, which we do not want.

```
func heal(amount):
    health += amount
```

What happens to health when we damage our character?

Suppose our character has `100` health.

Click here to access the app: Learn GDScript From Zero app [https://gdquest.github.io/learn-gdscript/?ref=godot-docs]

This app is an open-source project. To report bugs or contribute, head to the app's source code repository: GitHub repository [https://github.com/GDQuest/learn-gdscript].

In the next part, you will get an overview of the engine's essential concepts.

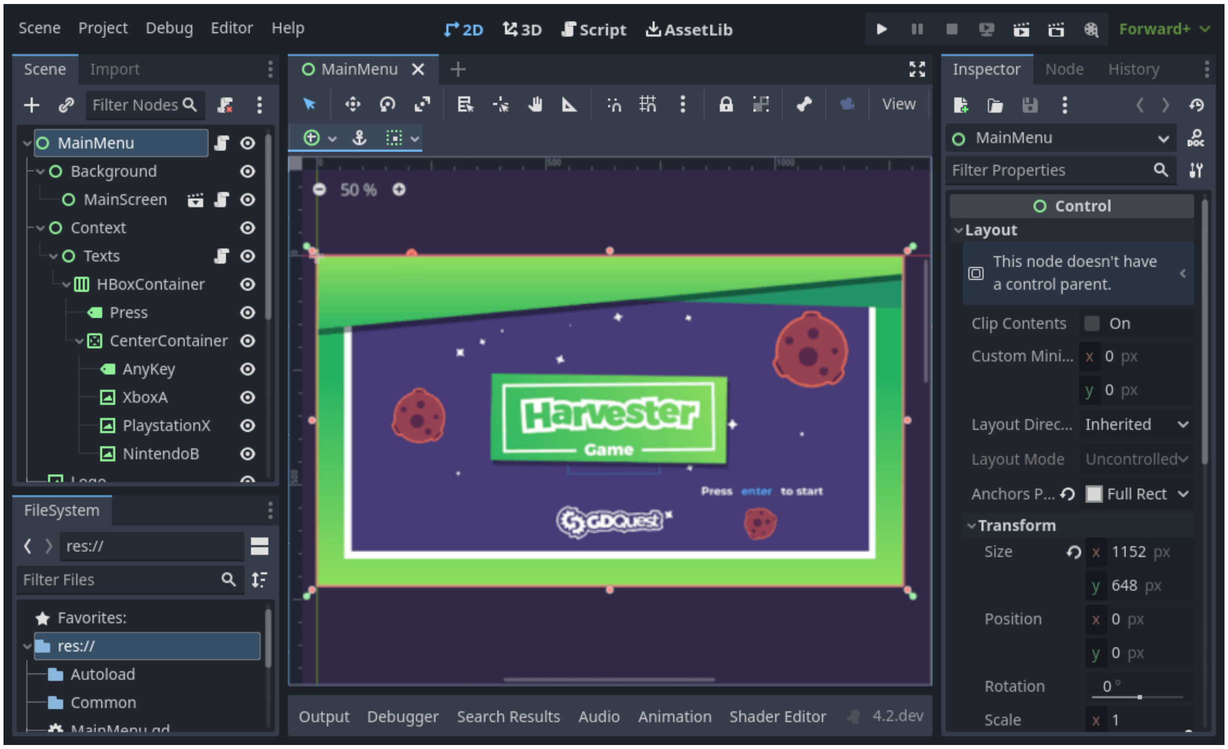
Overview of Godot's key concepts

Every game engine revolves around abstractions you use to build your applications. In Godot, a game is a **tree** of **nodes** that you group together into **scenes**. You can then wire these nodes so they can communicate using **signals**.

These are the four concepts you will learn here. We're going to look at them briefly to give you a sense of how the engine works. In the getting started series, you will get to use them in practice.

Scenes

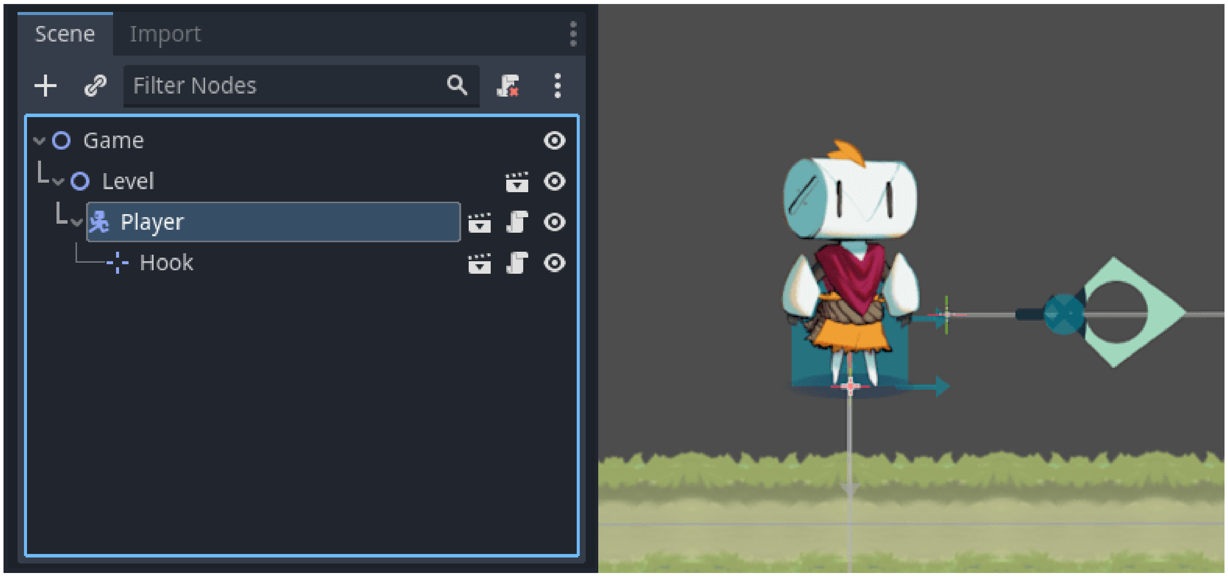
In Godot, you break down your game in reusable scenes. A scene can be a character, a weapon, a menu in the user interface, a single house, an entire level, or anything you can think of. Godot's scenes are flexible; they fill the role of both prefabs and scenes in some other game engines.


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Forward+
Scene
Import
Filter Nodes
MainMenu
Background
MainScreen
Context
Texts
HBoxContainer
Press
CenterContainer
AnyKey
XboxA
PlaystationX
NintendoB
Logo
FileSystem
res://
Filter Files
Favorites:
res://
Autoload
Common
MainMenu.gd
MainMenu
View
50 %
Harvester
Game
Press enter to start
GDQuest
Output
Debugger
Search Results
Audio
Animation
Shader Editor
4.2.dev
Inspector
Node
History
MainMenu
Filter Properties
Control
Layout
Transform

This node doesn't have a control parent.

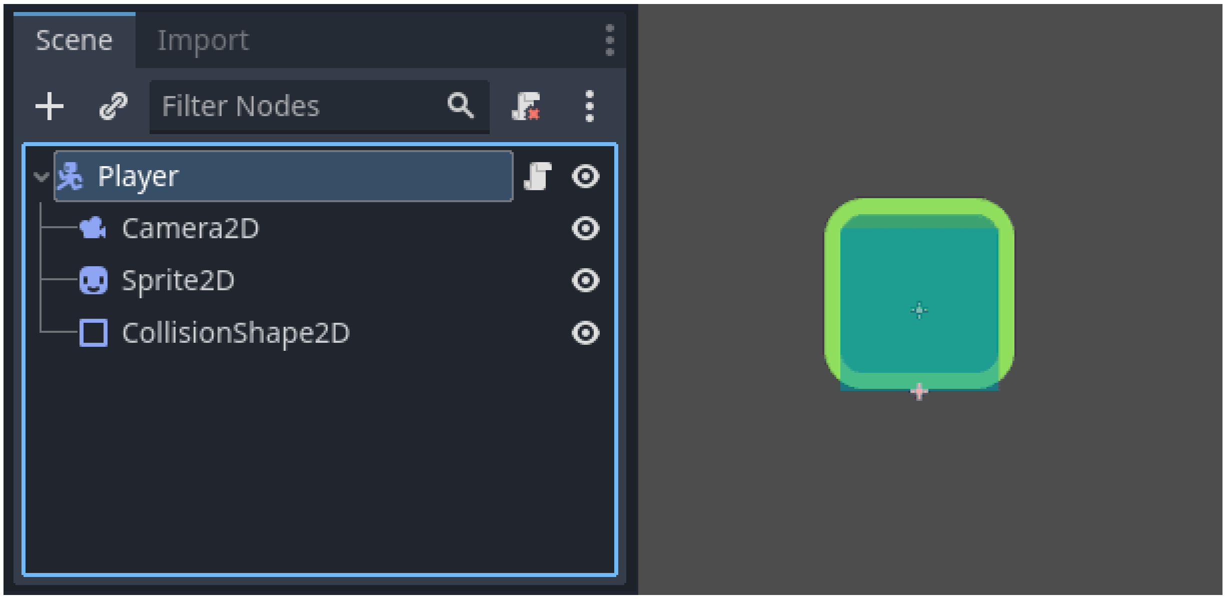
Clip Contents: On
Custom Mini...: x 0 px, y 0 px
Layout Direc...: Inherited
Layout Mode: Uncontrolled
Anchors P...: Full Rect
Size: x 1152 px, y 648 px
Position: x 0 px, y 0 px
Rotation: 0 °
Scale: x 1

You can also nest scenes. For example, you can put your character in a level, and drag and drop a scene as a child of it.


Scene
Import
Filter Nodes
Game
Level
Player
Hook

Nodes

A scene is composed of one or more **nodes**. Nodes are your game's smallest building blocks that you arrange into trees. Here's an example of a character's nodes.



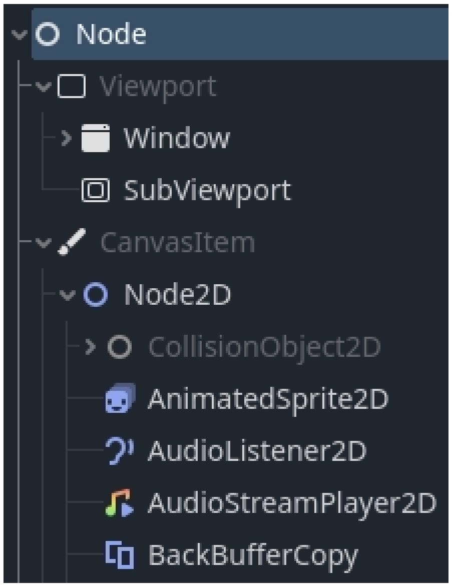
It is made of a `CharacterBody2D` node named "Player", a `Camera2D`, a `Sprite2D`, and a `CollisionShape2D`.

Note

The node names end with "2D" because this is a 2D scene. Their 3D counterparts have names that end with "3D". Be aware that "Spatial" Nodes are now called "Node3D" starting with Godot 4.

Notice how nodes and scenes look the same in the editor. When you save a tree of nodes as a scene, it then shows as a single node, with its internal structure hidden in the editor.

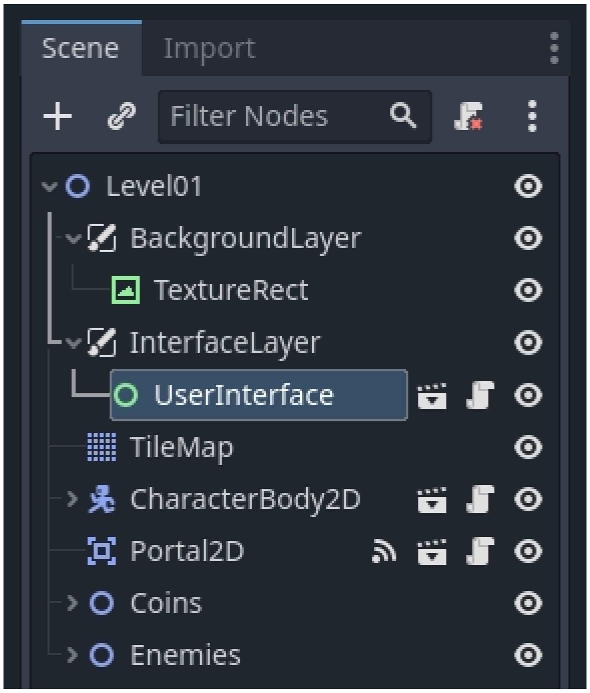
Godot provides an extensive library of base node types you can combine and extend to build more powerful ones. 2D, 3D, or user interface, you will do most things with these nodes.



Node
Viewport
Window
SubViewport
CanvasItem
Node2D
CollisionObject2D
AnimatedSprite2D
AudioListener2D
AudioStreamPlayer2D
BackBufferCopy

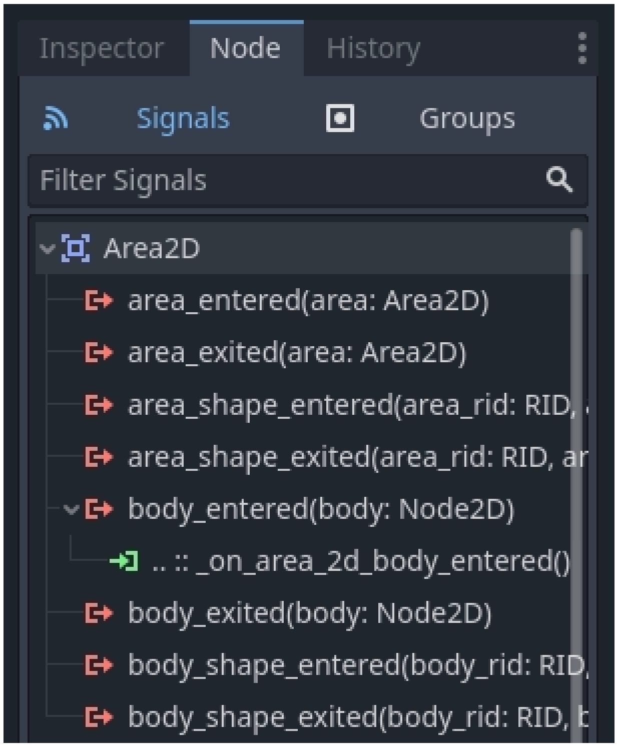
The scene tree

All your game's scenes come together in the **scene tree**, literally a tree of scenes. And as scenes are trees of nodes, the scene tree also is a tree of nodes. But it's easier to think of your game in terms of scenes as they can represent characters, weapons, doors, or your user interface.


Scene
Import
Filter Nodes
Level01
BackgroundLayer
TextureRect
InterfaceLayer
UserInterface
TileMap
CharacterBody2D
Portal2D
Coins
Enemies

Signals

Nodes emit signals when some event occurs. This feature allows you to make nodes communicate without hard-wiring them in code. It gives you a lot of flexibility in how you structure your scenes.


Inspector
Node
History
Signals
Groups
Filter Signals
Area2D
area_entered(area: Area2D)
area_exited(area: Area2D)
area_shape_entered(area_rid: RID,
area_shape_exited(area_rid: RID, ar
body_entered(body: Node2D)
.. :: _on_area_2d_body_entered()
body_exited(body: Node2D)
body_shape_entered(body_rid: RID,
body_shape_exited(body_rid: RID,

Note

Signals are Godot's version of the *observer* pattern. You can read more about it here: https://gameprogrammingpatterns.com/observer.html

For example, buttons emit a signal when pressed. You can connect to this signal to run code in reaction to this event, like starting the game or opening a menu.

Other built-in signals can tell you when two objects collided, when a character or monster entered a given area, and much more. You can also define new signals tailored to your game.

Summary

Nodes, scenes, the scene tree, and signals are four core concepts in Godot that you will manipulate all the time.

Nodes are your game's smallest building blocks. You combine them to create scenes that you then combine and nest into the scene tree. You can then use signals to make nodes react to events in other nodes or different scene tree branches.

After this short breakdown, you probably have many questions. Bear with us as you will get many answers throughout the getting started series.

First look at Godot's interface

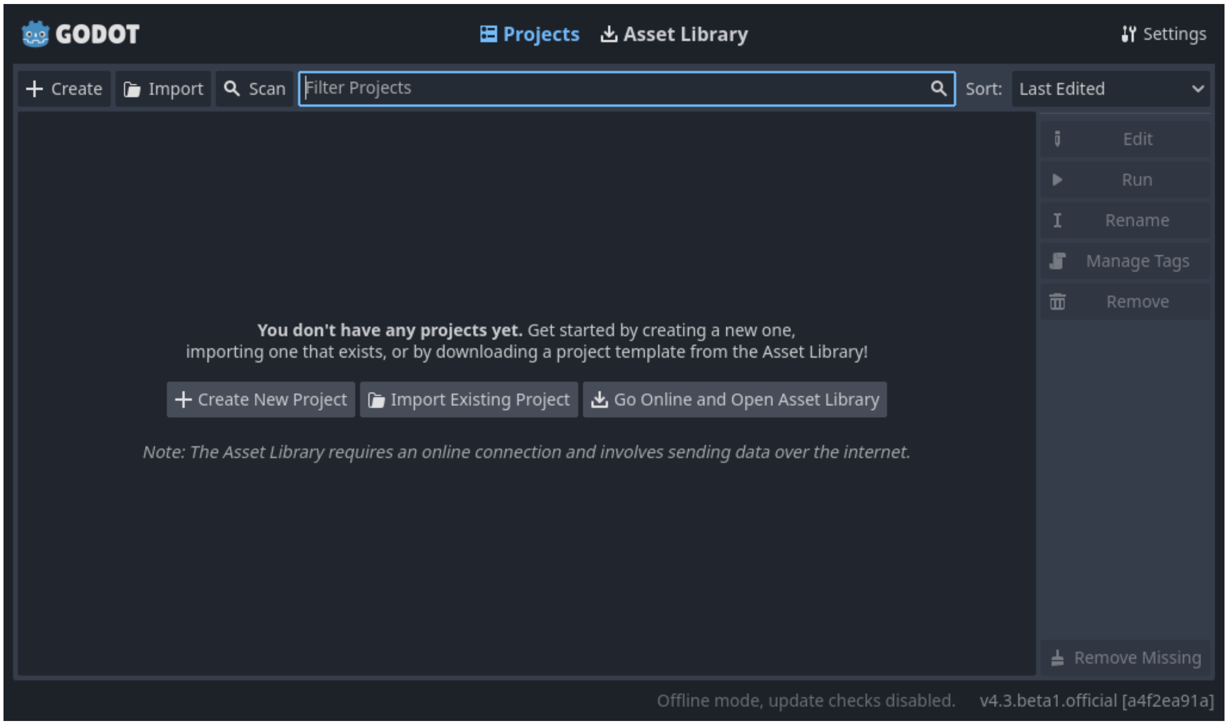
This page will give you a brief overview of Godot's interface. We're going to look at the different main screens and docks to help you situate yourself.

See also

For a comprehensive breakdown of the editor's interface and how to use it, see the Editor manual.

The Project Manager

When you launch Godot, the first window you see is the Project Manager. In the default tab **Projects**, you can manage existing projects, import or create new ones, and more.


GODOT
Projects
Asset Library
Settings
Create
Import
Scan
Filter Projects
Sort:
Last Edited
Edit
Run
Rename
Manage Tags
Remove
Create New Project
Import Existing Project
Go Online and Open Asset Library
Remove Missing

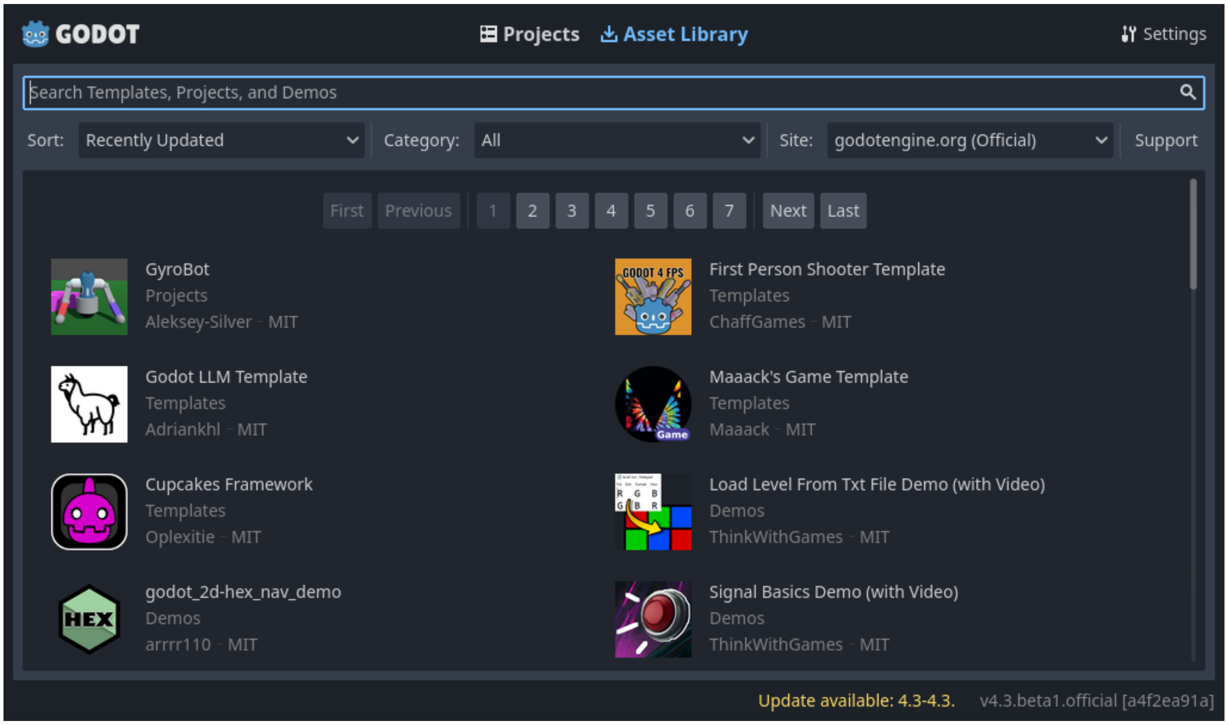
You don't have any projects yet. Get started by creating a new one,
importing one that exists, or by downloading a project template from the Asset Library!

Note: The Asset Library requires an online connection and involves sending data over the internet.

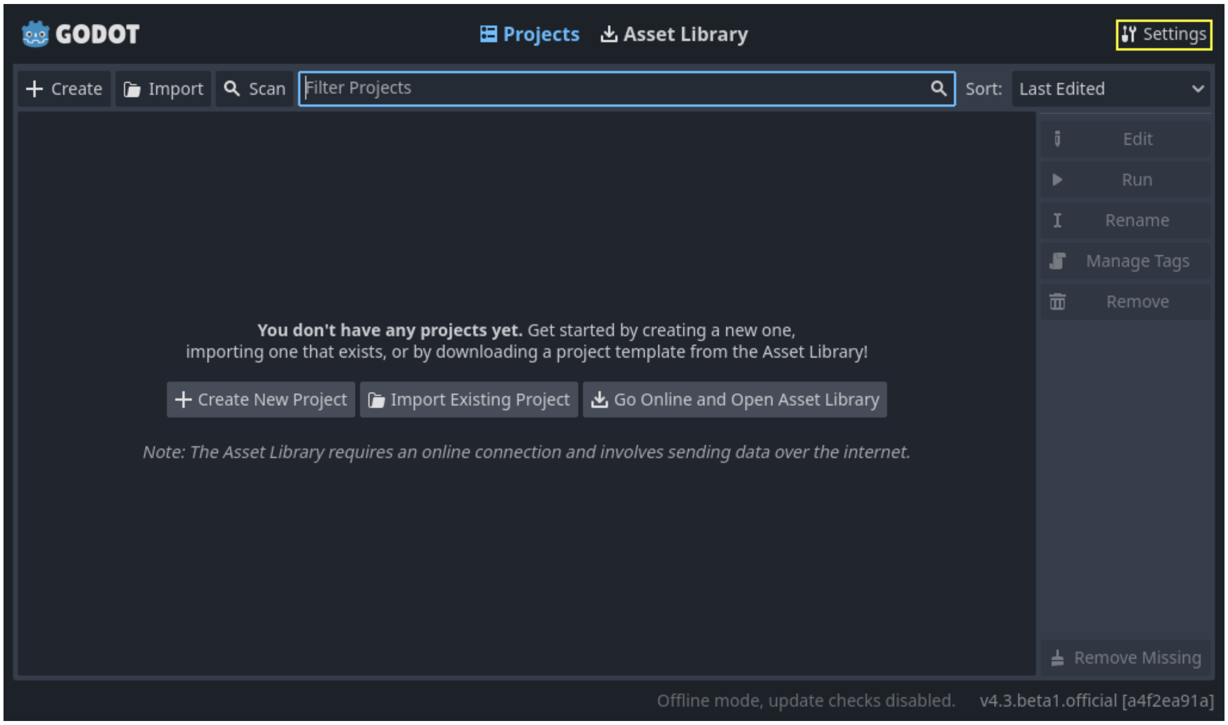
Offline mode, update checks disabled. v4.3.beta1.official [a4f2ea91a]

At the top of the window, there is another tab named **Asset Library**. The first time you go to this tab you'll see a "Go Online" button. For privacy reasons, the Godot project manager does not access the internet by default. To change this click the "Go Online" button. You can change this option later in the settings.

Once your network mode is set to "online", you can search for demo projects in the open source asset library, which includes many projects developed by the community:


GODOT
Projects
Asset Library
Settings
Search Templates, Projects, and Demos
Sort: Recently Updated
Category: All
Site: godotengine.org (Official)
Support
First
Previous
1
2
3
4
5
6
7
Next
Last
GyroBot
Projects
Aleksey-Silver MIT
First Person Shooter Template
Templates
ChaffGames MIT
Godot LLM Template
Templates
Adriankhl MIT
Maaack's Game Template
Templates
Maaack MIT
Cupcakes Framework
Templates
Oplexitie MIT
Load Level From Txt File Demo (with Video)
Demos
ThinkWithGames MIT
godot_2d-hex_nav_demo
Demos
arrrr110 MIT
Signal Basics Demo (with Video)
Demos
ThinkWithGames MIT
Update available: 4.3-4.3.
v4.3.beta1.official [a4f2ea91a]

The Project Manager's settings can be opened using the **Settings** menu:


GODOT
Projects
Asset Library
Settings
Create
Import
Scan
Filter Projects
Sort: Last Edited
Edit
Run
Rename
Manage Tags
Remove
Create New Project
Import Existing Project
Go Online and Open Asset Library
Remove Missing
Offline mode, update checks disabled.
v4.3.beta1.official [a4f2ea91a]

You don't have any projects yet. Get started by creating a new one, importing one that exists, or by downloading a project template from the Asset Library!

Note: The Asset Library requires an online connection and involves sending data over the internet.

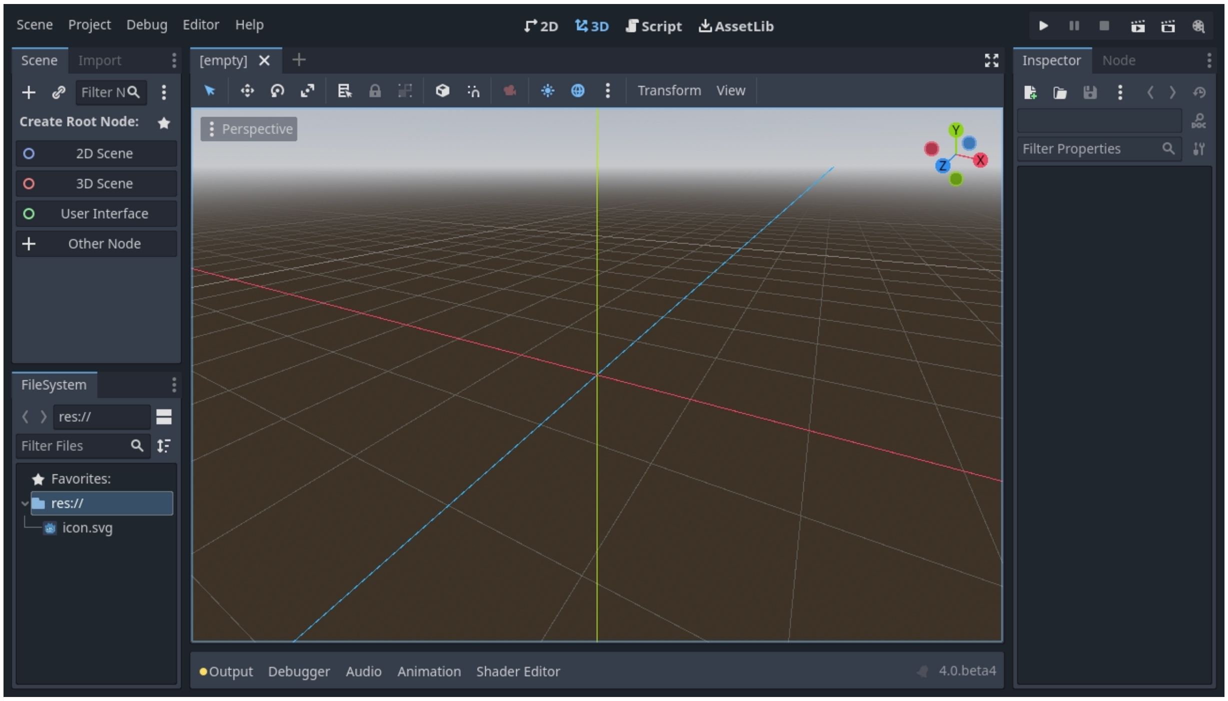
From here, you can change the editor's language (default is the system language), interface theme, display scale, network mode, and also the directory naming convention.

See also

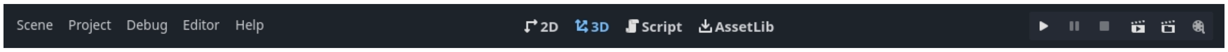
To learn the Project Manager's ins and outs, read Using the Project Manager.

First look at Godot's editor

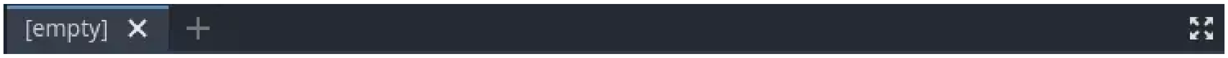
When you open a new or an existing project, the editor's interface appears. Let's look at its main areas:


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Scene
Import
Filter N
Create Root Node:
2D Scene
3D Scene
User Interface
Other Node
[empty]
Transform
View
Perspective
Y
X
Z
Inspector
Node
Filter Properties
FileSystem
res://
Filter Files
Favorites:
res://
icon.svg
Output
Debugger
Audio
Animation
Shader Editor
4.0.beta4

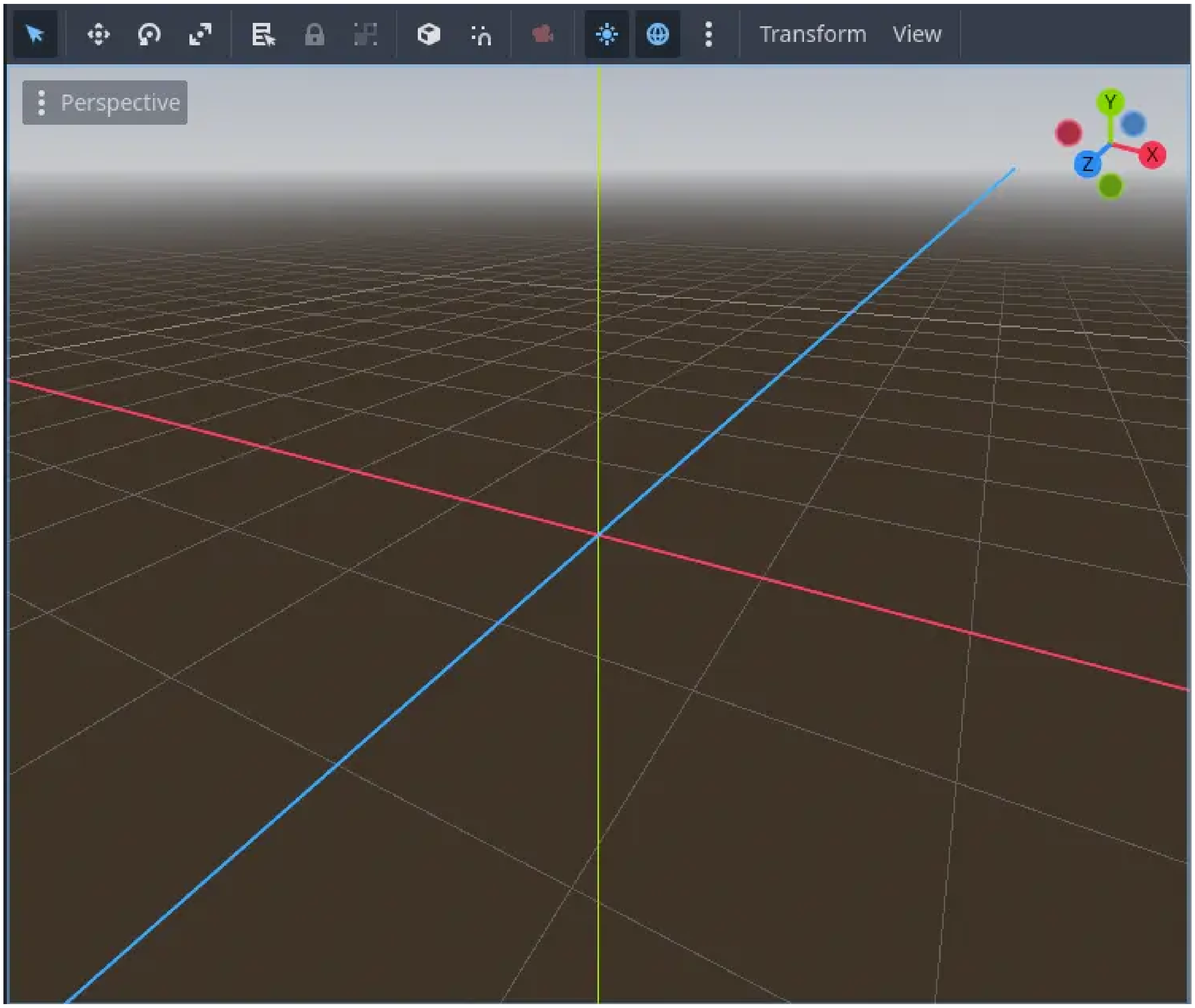
By default, along the window's top edge, it features **main menu** on the left, **workspace** switching buttons in the center (active workspace is highlighted), and **playtest** buttons on the right:


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib

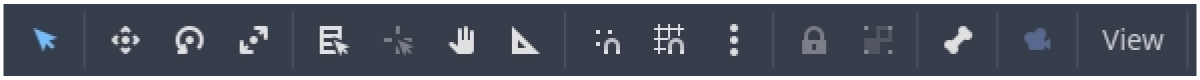
Just below the workspace buttons, the opened scenes as tabs are seen. The plus (+) button right next to the tabs will add a new scene to the project. With the button on the far right, distraction-free mode can be toggled, which maximizes or restores the **viewport**'s size by hiding **docks** in the interface:



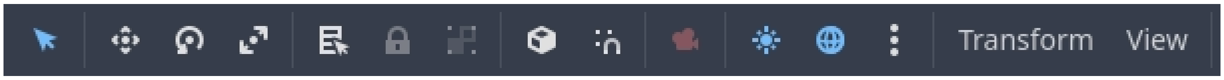
In the center, below the scene selector is the **viewport** with its **toolbar** at the top, where you'll find different tools to move, scale, or lock the scene's nodes (currently the 3D workspace is active):


Transform
View
Perspective
Y
X
Z

This toolbar changes based on the context and selected node. Here is the 2D toolbar:



Below is the 3D one:



See also

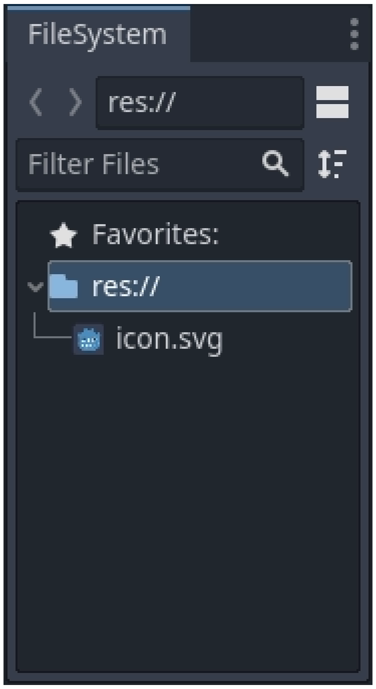
To learn more on workspaces, read The four main screens.

See also

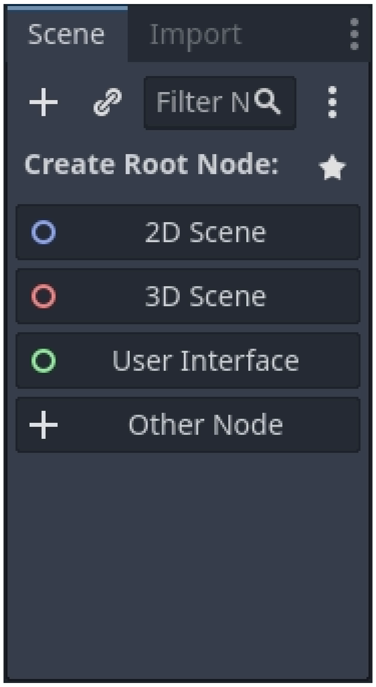
To learn more on the 3D viewport and 3D in general, read Introduction to 3D.

On either side of the viewport sit the **docks**. And at the bottom of the window lies the **bottom panel**.

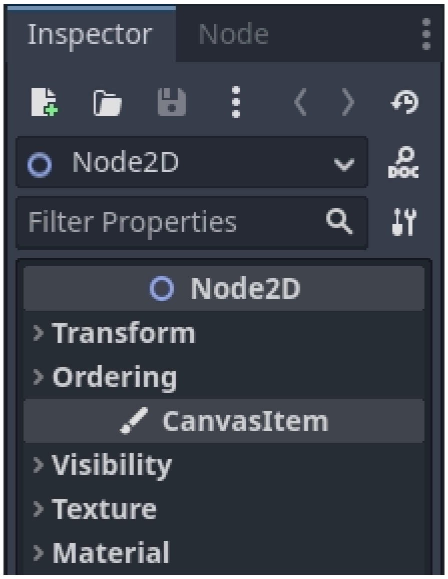
Let's look at the docks. The **FileSystem** dock lists your project files, including scripts, images, audio samples, and more:


FileSystem
res://
Filter Files
Favorites:
res://
icon.svg

The **Scene** dock lists the active scene's nodes:


Scene
Import
Filter N
Create Root Node:
2D Scene
3D Scene
User Interface
Other Node

The **Inspector** allows you to edit the properties of a selected node:


Inspector
Node
Node2D
Filter Properties
Node2D
Transform
Ordering
CanvasItem
Visibility
Texture
Material

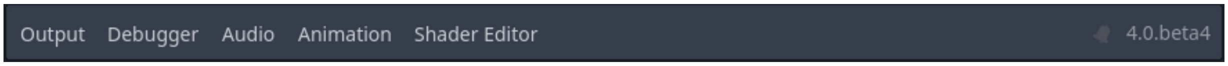
See also

To read more on inspector, see The Inspector.

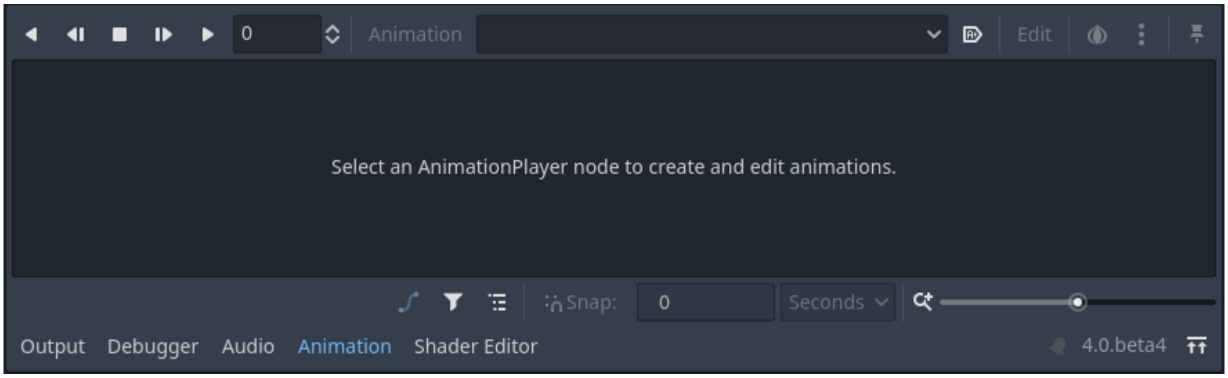
See also

Docks can be customized. Read more on Moving and resizing docks.

The **bottom panel**, situated below the viewport, is the host for the debug console, the animation editor, the audio mixer, and more. They can take precious space, that's why they're folded by default:


Output
Debugger
Audio
Animation
Shader Editor
4.0.beta4

When you click on one, it expands vertically. Below, you can see the animation editor opened:


0
Animation
Edit
Snap:
0
Seconds
Output
Debugger
Audio
Animation
Shader Editor
4.0.beta4

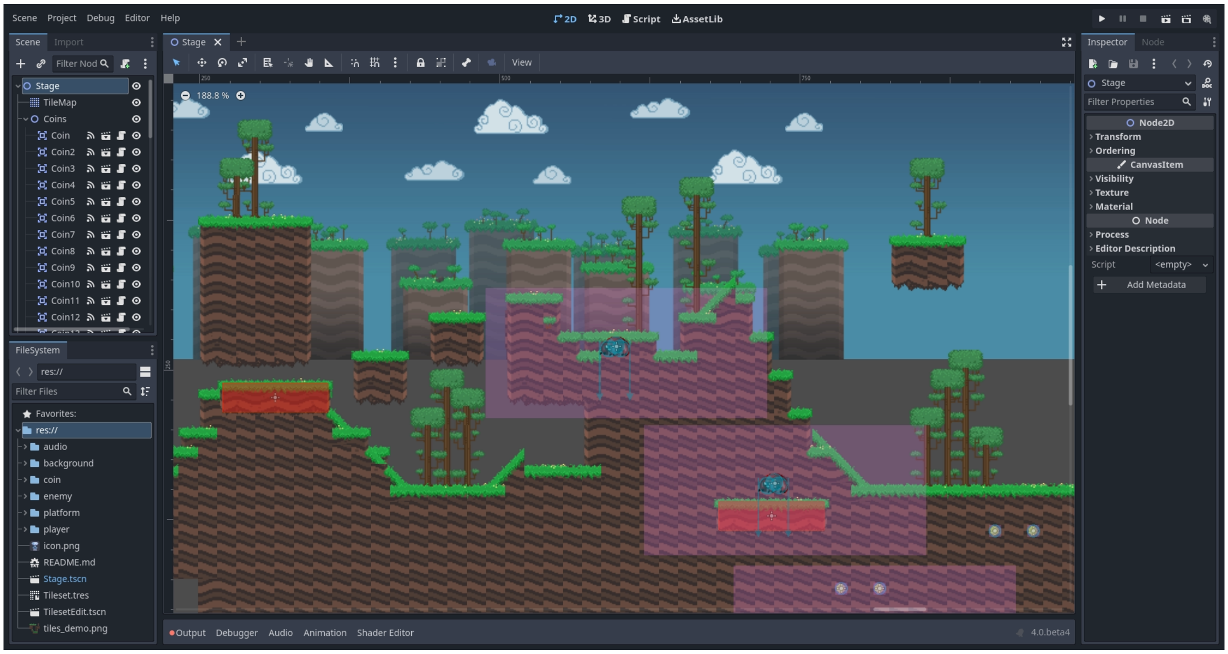
Select an AnimationPlayer node to create and edit animations.

Bottom panels can also be shown or hidden using the shortcuts defined in **Editor Settings > Shortcuts**, under the **Bottom Panels** category.

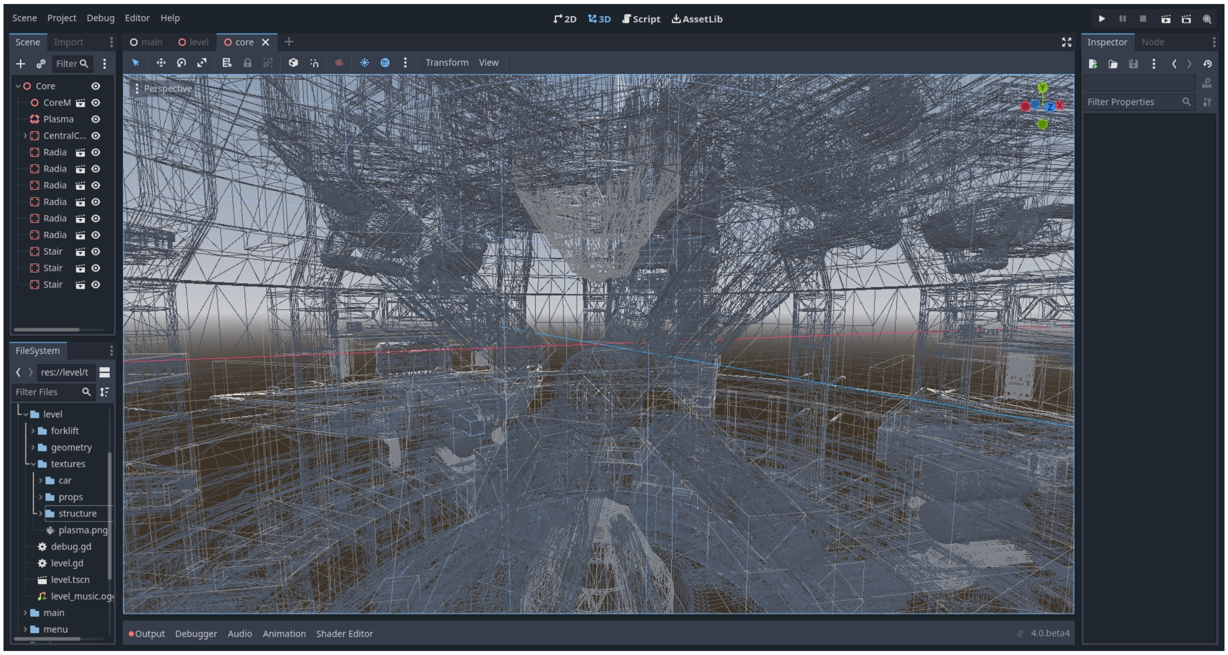
The four main screens

There are four main screen buttons centered at the top of the editor: 2D, 3D, Script, and Asset Library.

You'll use the **2D screen** for all types of games. In addition to 2D games, the 2D screen is where you'll build your interfaces.


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Scene
Import
Stage
TileMap
Coins
Coin
Coin2
Coin3
Coin4
Coin5
Coin6
Coin7
Coin8
Coin9
Coin10
Coin11
Coin12
FileSystem
res://
Filter Files
Favorites:
res://
audio
background
coin
enemy
platform
player
icon.png
README.md
Stage.tscn
Tileset.tres
TilesetEdit.tscn
tiles_demo.png
Stage
View
188.8 %
Output
Debugger
Audio
Animation
Shader Editor
4.0.beta4
Inspector
Node
Stage
Filter Properties
Node2D
Transform
Ordering
CanvasItem
Visibility
Texture
Material
Node
Process
Editor Description
Script
<empty>
Add Metadata

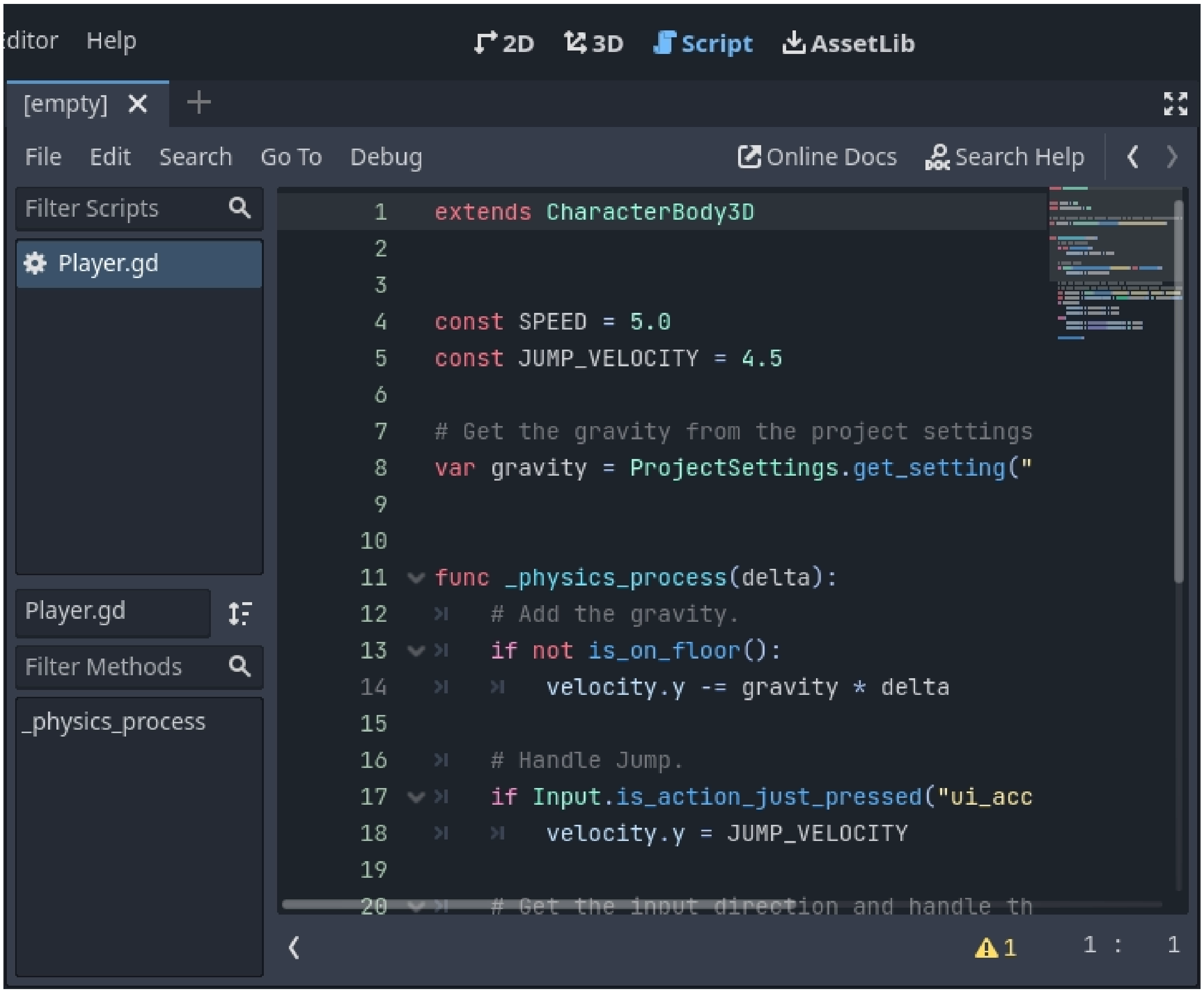
In the **3D screen**, you can work with meshes, lights, and design levels for 3D games.


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Scene
Import
main
level
core
Transform
View
Perspective
Inspector
Node
Filter Properties
Core
CoreM
Plasma
CentralC...
Radia
Radia
Radia
Radia
Radia
Radia
Stair
Stair
Stair
FileSystem
res://level/t
Filter Files
level
forklift
geometry
textures
car
props
structure
plasma.png
debug.gd
level.gd
level.tscn
level_music.ogg
main
menu
Output
Debugger
Audio
Animation
Shader Editor
4.0.beta4

Note

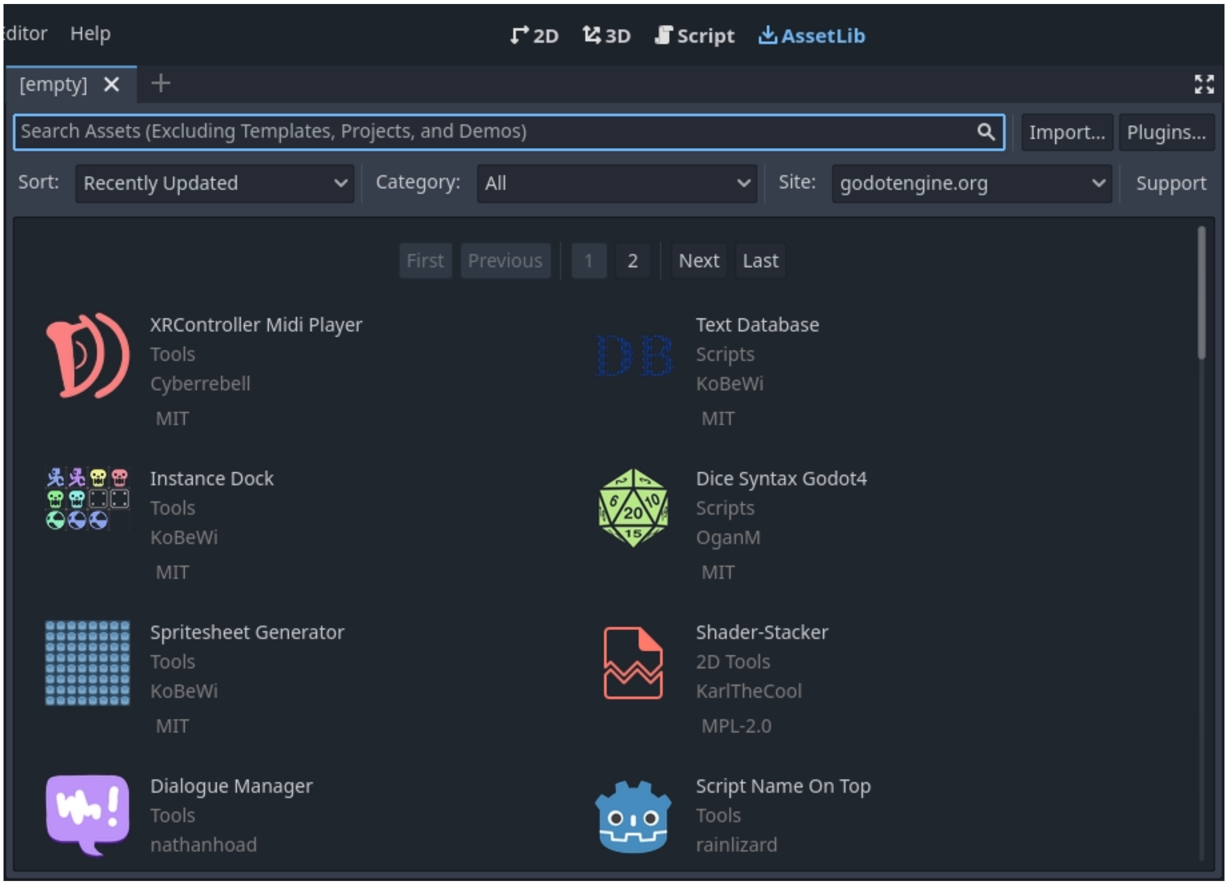
Read Introduction to 3D for more detail about the **3D main screen**.

The **Script screen** is a complete code editor with a debugger, rich auto-completion, and built-in code reference.


ditor
Help
2D
3D
Script
AssetLib
[empty]
File
Edit
Search
Go To
Debug
Online Docs
Search Help
Filter Scripts
Player.gd
Player.gd
Filter Methods
_physics_process
⚠1
1 : 1

```
extends CharacterBody3D


const SPEED = 5.0
const JUMP_VELOCITY = 4.5

# Get the gravity from the project settings
var gravity = ProjectSettings.get_setting("


func _physics_process(delta):
    # Add the gravity.
    if not is_on_floor():
        velocity.y -= gravity * delta

    # Handle Jump.
    if Input.is_action_just_pressed("ui_acc
        velocity.y = JUMP_VELOCITY

    # Get the input direction and handle th
```

Finally, the **Asset Library** is a library of free and open source add-ons, scripts, and assets to use in your projects.


ditor Help
2D
3D
Script
AssetLib
[empty]
Search Assets (Excluding Templates, Projects, and Demos)
Import...
Plugins...
Sort: Recently Updated
Category: All
Site: godotengine.org
Support
First Previous 1 2 Next Last
XRController Midi Player
Tools
Cyberrebell
MIT
Text Database
Scripts
KoBeWi
MIT
Instance Dock
Tools
KoBeWi
MIT
Dice Syntax Godot4
Scripts
OganM
MIT
Spritesheet Generator
Tools
KoBeWi
MIT
Shader-Stacker
2D Tools
KarlTheCool
MPL-2.0
Dialogue Manager
Tools
nathanhoad
Script Name On Top
Tools
rainlizard

See also

You can learn more about the asset library in [About the Asset Library]().

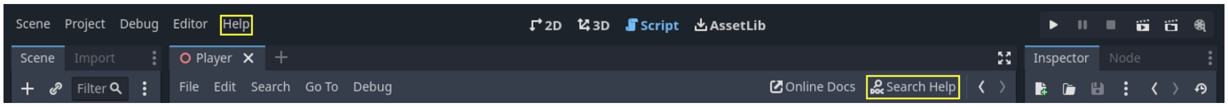
Integrated class reference

Godot comes with a built-in class reference.

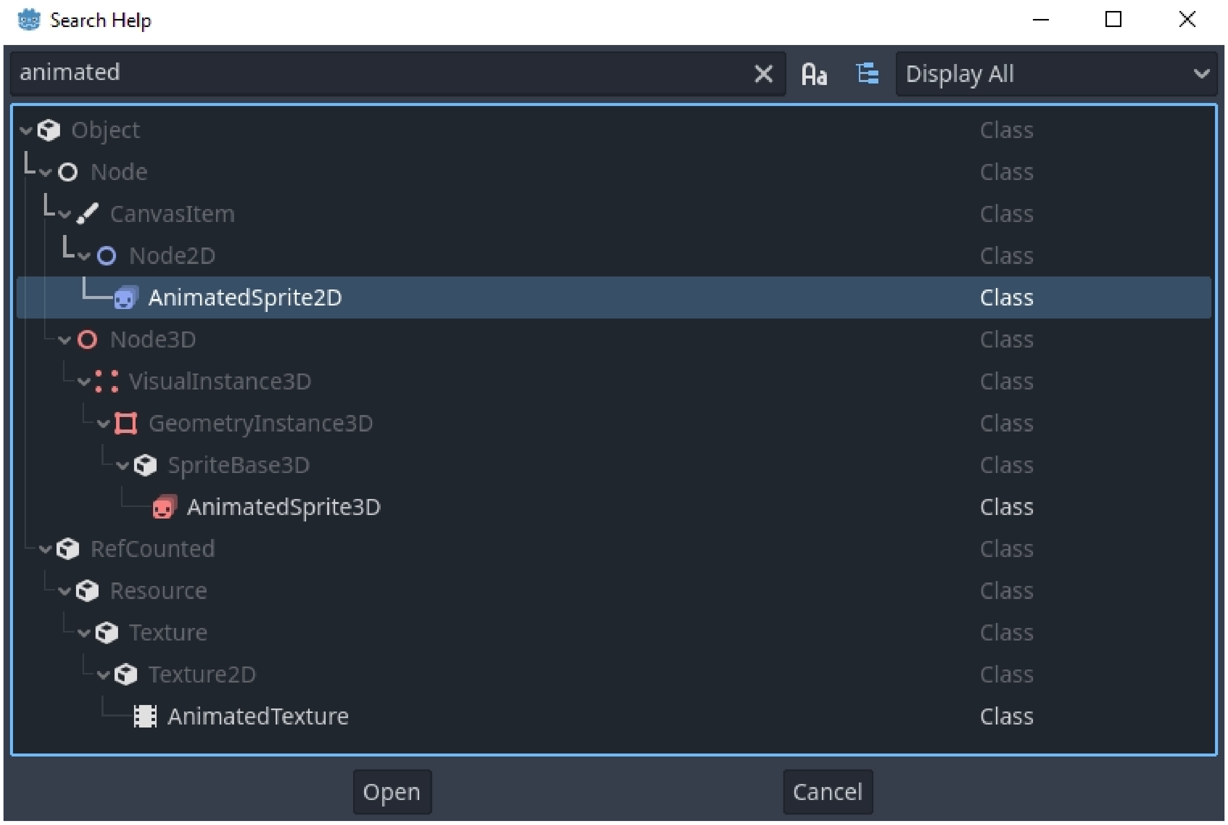
You can search for information about a class, method, property, constant, or signal by any one of the following methods:

- Pressing `F1` (or `Opt + Space` on macOS, or `Fn + F1` for laptops with a `Fn` key) anywhere in the editor.

- Clicking the "Search Help" button in the top-right of the Script main screen.
- Clicking on the Help menu and Search Help.
- `Ctrl + Click` (`Cmd + Click` on macOS) on a class name, function name, or built-in variable in the script editor.


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Scene
Import
Player
Online Docs
Search Help
Inspector
Node
Filter
File
Edit
Search
Go To
Debug

When you do any of these, a window pops up. Type to search for any item. You can also use it to browse available objects and methods.

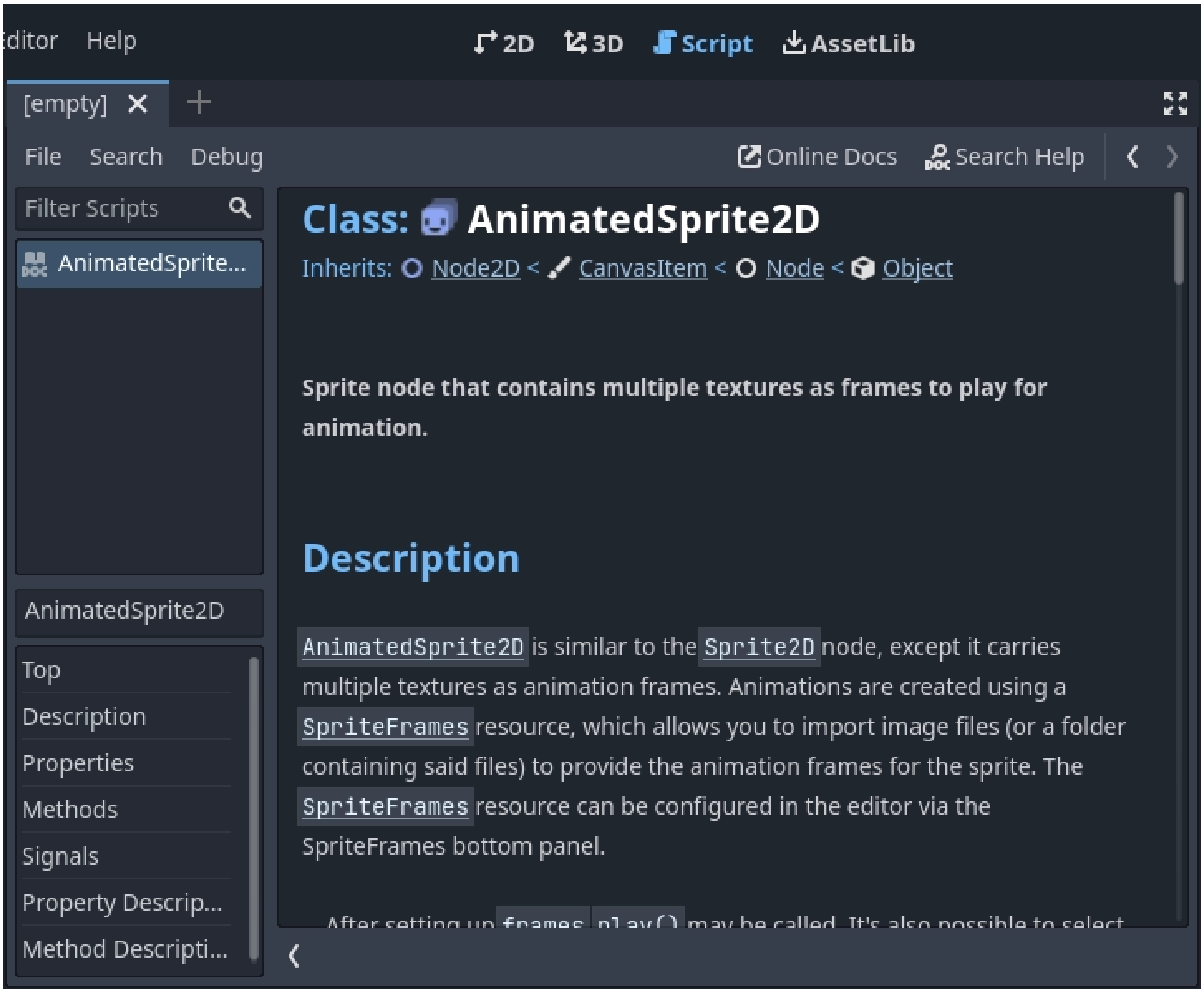

Search Help
Open
Cancel

animated

Display All

Object	Class
Node	Class
CanvasItem	Class
Node2D	Class
AnimatedSprite2D	Class
Node3D	Class
VisualInstance3D	Class
GeometryInstance3D	Class
SpriteBase3D	Class
AnimatedSprite3D	Class
RefCounted	Class
Resource	Class
Texture	Class
Texture2D	Class
AnimatedTexture	Class

Double-click on an item to open the corresponding page in the script main screen.


ditor
Help
2D
3D
Script
AssetLib
[empty]
File
Search
Debug
Online Docs
Search Help
Filter Scripts
AnimatedSprite...
AnimatedSprite2D
Top
Description
Properties
Methods
Signals
Property Descrip...
Method Descripti...

Class: AnimatedSprite2D

Inherits: Node2D < CanvasItem < Node < Object

Sprite node that contains multiple textures as frames to play for animation.

Description

`AnimatedSprite2D` is similar to the `Sprite2D` node, except it carries multiple textures as animation frames. Animations are created using a `SpriteFrames` resource, which allows you to import image files (or a folder containing said files) to provide the animation frames for the sprite. The `SpriteFrames` resource can be configured in the editor via the SpriteFrames bottom panel.

After setting up `frames`, `play()` may be called. It's also possible to select

Alternatively,

- Clicking while pressing the `Ctrl` key on a class name, function name, or built-in variable in the script editor.
- Right-clicking on nodes and choosing **Open Documentation** or choosing **Lookup Symbol** for elements in script editor will directly open their documentation.

Learning new features

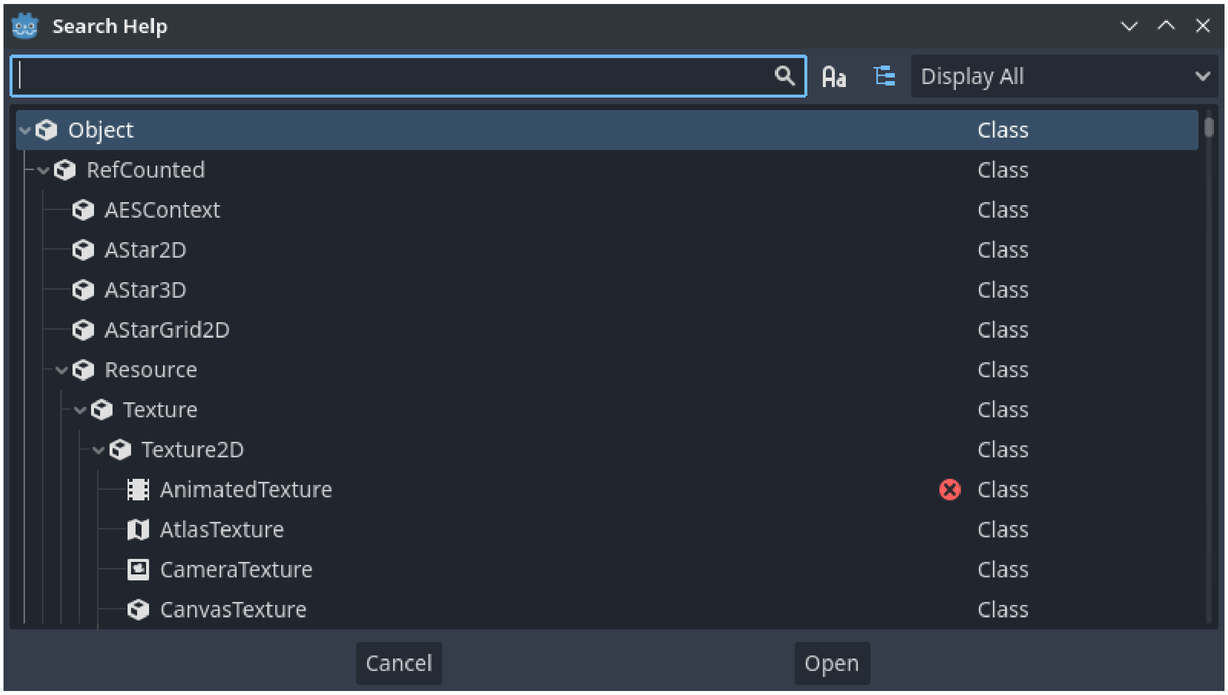
Godot is a feature-rich game engine. There is a lot to learn about it. This page explains how you can use the online manual, built-in code reference, and join online communities to learn new features and techniques.

Making the most of this manual

What you are reading now is the user manual. It documents each of the engine's concepts and available features. When learning a new topic, you can start by browsing the corresponding section of this website. The left menu allows you to explore broad topics while the search bar will help you find more specific pages. If a page exists for a given theme, it will often link to more related content.

Search docs

The manual has a companion class reference that explains each Godot class's available functions and properties when programming. While the manual covers general features, concepts, and how to use the editor, the reference is all about using Godot's scripting API (Application Programming Interface). You can access it both online and offline. We recommend browsing the reference offline, from within the Godot editor. To do so, go to Help -> Search Help or press `F1`.

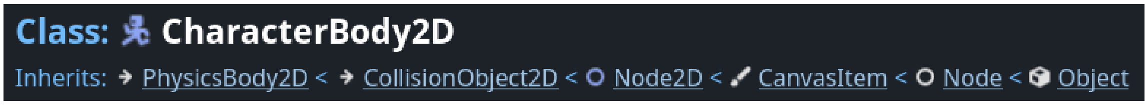


Object	Class
RefCounted	Class
AESContext	Class
AStar2D	Class
AStar3D	Class
AStarGrid2D	Class
Resource	Class
Texture	Class
Texture2D	Class
AnimatedTexture	☒ Class
AtlasTexture	Class
CameraTexture	Class
CanvasTexture	Class

To browse it online, head to the manual's Class Reference section.

A class reference's page tells you:

1. Where the class exists in the inheritance hierarchy. You can click the top links to jump to parent classes and see the properties and methods a type inherits.



Class: CharacterBody2D

Inherits: PhysicsBody2D < CollisionObject2D < Node2D < CanvasItem < Node < Object

2. A summary of the class's role and use cases.

3. An explanation of the class's properties, methods, signals, enums, and constants.

4. Links to manual pages further detailing the class.

Note

If the manual or class reference is missing or has insufficient information, please open an Issue in the official godot-docs [https://github.com/godotengine/godot-docs/issues] GitHub repository to report it.

You can hold `Ctrl` (macOS `Cmd`) and then mouseover text like the name of a class, property, method, signal, or constant to underline it, then `Ctrl + Click` (macOS `Cmd + Click`) it to jump to it.

Learning to think like a programmer

Teaching programming foundations and how to think like a game developer is beyond the scope of Godot's documentation. If you're new to programming, we recommend two excellent free resources to get you started:

1. Harvard university offers a free courseware to learn to program, CS50 [https://cs50.harvard.edu/x/]. It will teach you programming fundamentals, how code works, and how to think like a programmer. These skills are essential to become a game developer and learn any game engine efficiently. You can see this course as an investment that will save you time and trouble when you learn to create games.
2. If you prefer books, check out the free ebook Automate The Boring Stuff With Python [https://automatetheboringstuff.com/] by Al Sweigart.

Learning with the community

Godot has a growing community of users. If you're stuck on a problem or need help to better understand how to achieve something, you can ask other users for help on one of the many active communities [https://godotengine.org/community].

The best place to ask questions and find already answered ones is the official Questions & Answers [https://ask.godotengine.org/] site. These responses show up in search engine results and get saved, allowing other users to benefit from discussions on the platform. Once you have asked a question there, you can share its link on other social platforms. Before asking a question, be sure to look for existing answers that might solve your problem on this website or using your preferred search engine.

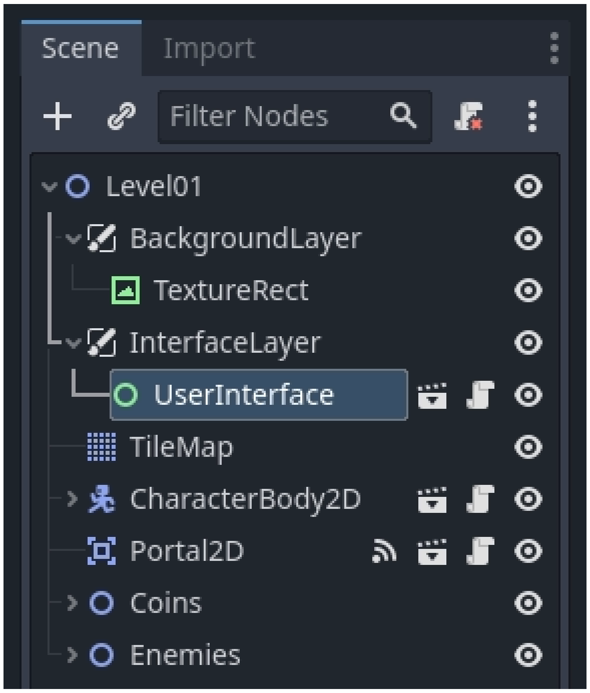
Asking questions well and providing details will help others answer you faster and better. When asking questions, we recommend including the following information:

1. **Describe your goal**. You want to explain what you are trying to achieve design-wise. If you are having trouble figuring out how to make a solution work, there may be a different, easier solution that accomplishes the same goal.

2. If there is an error involved, **share the exact error message**. You can copy the exact error message in the editor's Debugger bottom panel by clicking the Copy Error icon. Knowing what it says can help community members better identify how you triggered the error.

3. If there is code involved, **share a code sample**. Other users won't be able to help you fix a problem without seeing your code. Share the code as text directly. To do so, you can copy and paste a short code snippet in a chat box, or use a website like Pastebin [https://pastebin.com/] to share long files.

4. **Share a screenshot** of your *Scene* dock along with your written code. Most of the code you write affects nodes in your scenes. As a result, you should think of those scenes as part of your source code.


Scene
Import
Filter Nodes
Level01
BackgroundLayer
TextureRect
InterfaceLayer
UserInterface
TileMap
CharacterBody2D
Portal2D
Coins
Enemies

Also, please don't take a picture with your phone, the low quality and screen reflections can make it hard to understand the image. Your operating system should have a built-in tool to take screenshots with the `PrtSc` (Print Screen) key (macOS: use `Cmd + Shift + 3` for a full screen shot, more information here [https://support.apple.com/guide/mac-help/take-a-screenshot-mh26782/mac]).

Alternatively, you can use a program like ShareX [https://getsharex.com/] on Windows, or FlameShot [https://flameshot.org/] on Linux.

5. Sharing a video of your running game can also be really **useful to troubleshoot your game**. You can use programs like OBS Studio [https://obsproject.com/] and Screen to GIF [https://www.screentogif.com/] to capture your screen.

   You can then use a service like streamable [https://streamable.com/] or a cloud provider to upload and share your videos for free.

6. If you're not using the stable version of Godot, please mention the version you're using. The answer can be different as available features and the interface evolve rapidly.

Following these guidelines will maximize your chances of getting the answer you're looking for. They will save time both for you and the persons helping you.

Community tutorials

This manual aims to provide a comprehensive reference of Godot's features. Aside from the 2D and 3D getting started series, it does not contain tutorials to implement specific game genres. If you're looking for a tutorial about creating a role-playing game, a platformer, or other, please see Tutorials and resources, which lists content made by the Godot community.

Godot's design philosophy

Now that you've gotten your feet wet, let's talk about Godot's design.

Every game engine is different and fits different needs. Not only do they offer a range of features, but the design of each engine is unique. This leads to different workflows and different ways to form your games' structures. This all stems from their respective design philosophies.

This page is here to help you understand how Godot works, starting with some of its core pillars. It is not a list of available features, nor is it an engine comparison. To know if any engine can be a good fit for your project, you need to try it out for yourself and understand its design and limitations.

Please watch Godot explained in 7 minutes [https://www.youtube.com/watch?v=yS9cuu5o5Ug] if you're looking for an overview of the engine's features.

Object-oriented design and composition

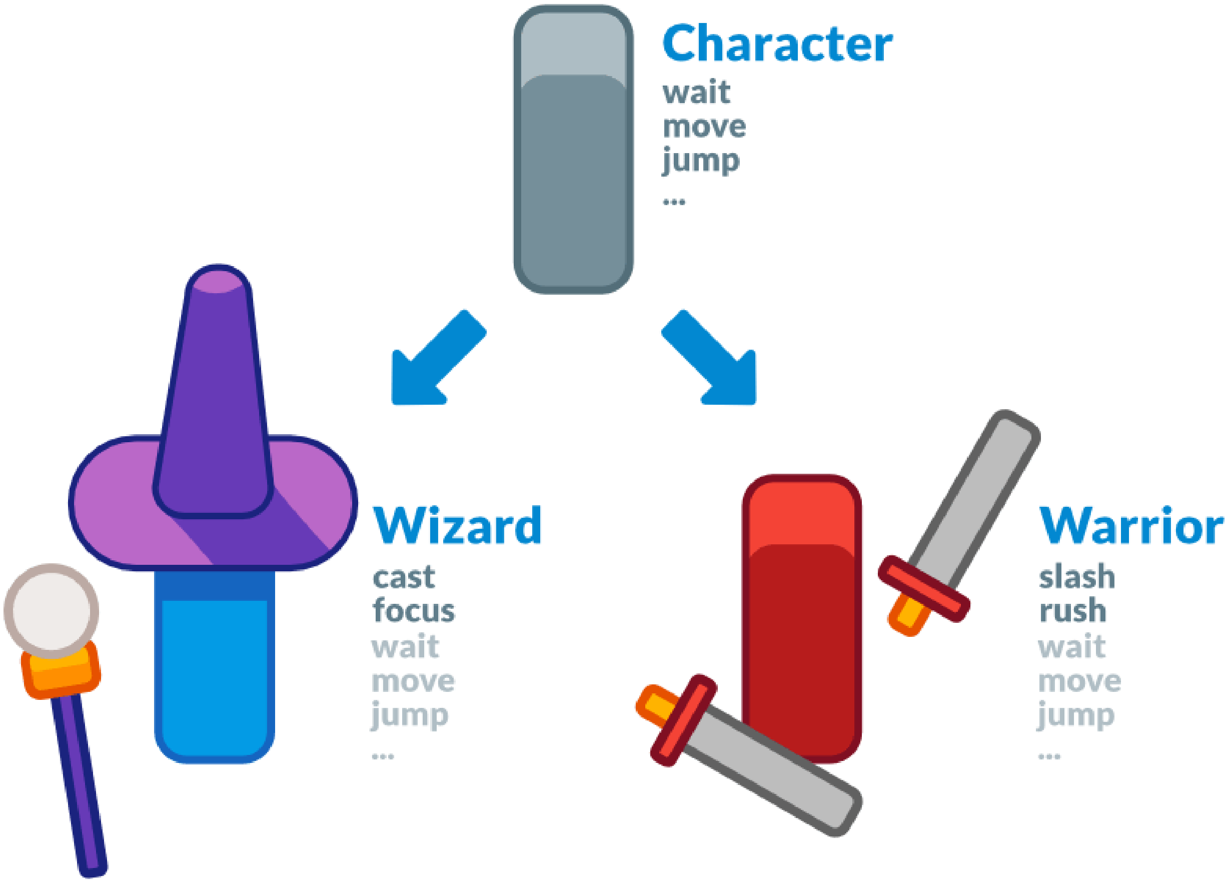
Godot embraces object-oriented design at its core with its flexible scene system and Node hierarchy. It tries to stay away from strict programming patterns to offer an intuitive way to structure your game.

For one, Godot lets you **compose or aggregate** scenes. It's like nested prefabs: you can create a BlinkingLight scene and a BrokenLantern scene that uses the BlinkingLight. Then, create a city filled with BrokenLanterns. Change the BlinkingLight's color, save, and all the BrokenLanterns in the city will update instantly.

On top of that, you can **inherit** from any scene.

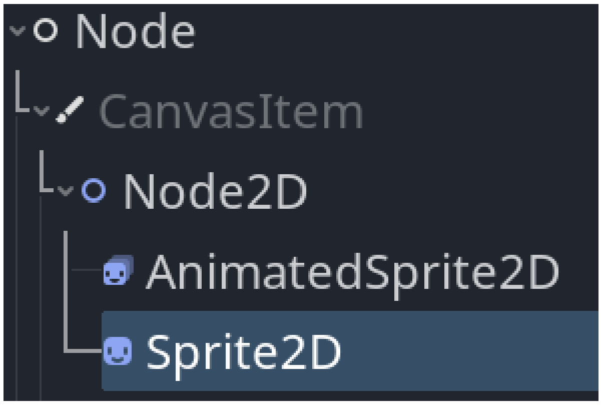
A Godot scene could be a Weapon, a Character, an Item, a Door, a Level, part of a level… anything you'd like. It works like a class in pure code, except you're free to design it by using the editor, using only the code, or mixing and matching the two.

It's different from prefabs you find in several 3D engines, as you can then inherit from and extend those scenes. You may create a Magician that extends your Character. Modify the Character in the editor and the Magician will update as well. It helps you build your projects so that their structure matches the game's design.


Character
wait
move
jump
...
Wizard
cast
focus
wait
move
jump
...
Warrior
slash
rush
wait
move
jump
...

Also note that Godot offers many different types of objects called nodes, each with a specific purpose. Nodes are part of a tree and always inherit from their parents up to the Node class. Although the engine does feature some nodes like collision shapes that a parent physics body will use, most nodes work independently from one another.

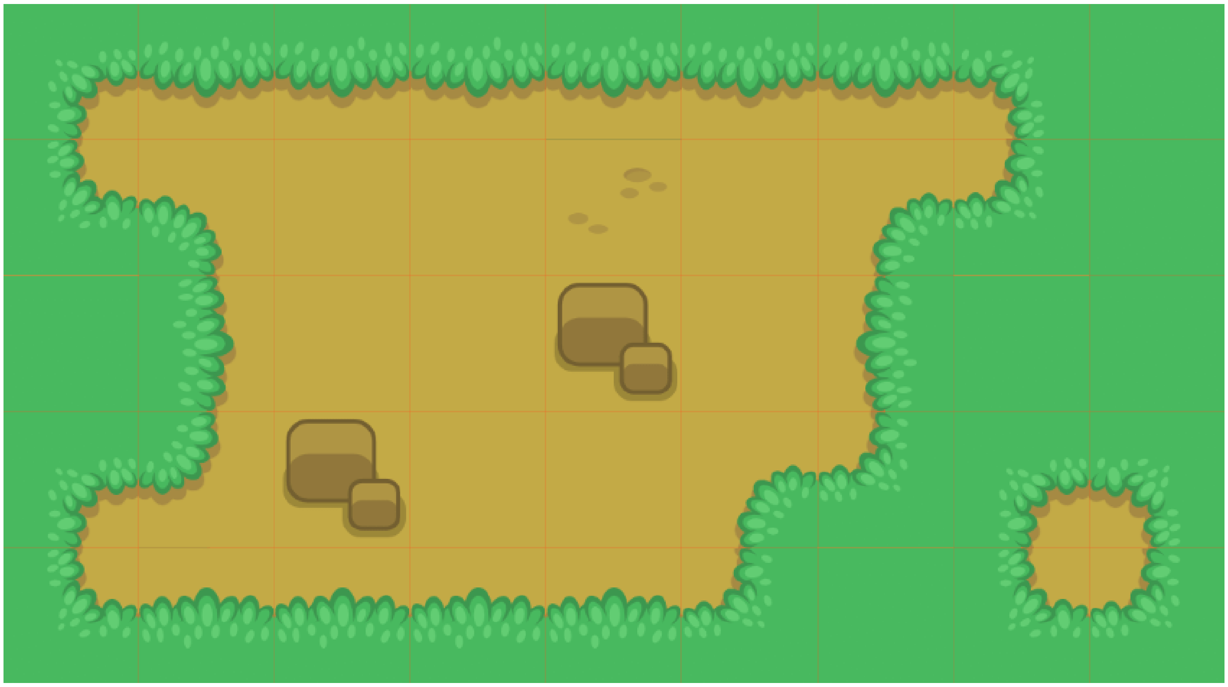
In other words, Godot's nodes do not work like components in some other game engines.



Sprite2D is a Node2D, a CanvasItem and a Node. It has all the properties and features of its three parent classes, like transforms or the ability to draw custom shapes and render with a custom shader.

All-inclusive package

Godot tries to provide its own tools to answer most common needs. It has a dedicated scripting workspace, an animation editor, a tilemap editor, a shader editor, a debugger, a profiler, the ability to hot-reload locally and on remote devices, etc.



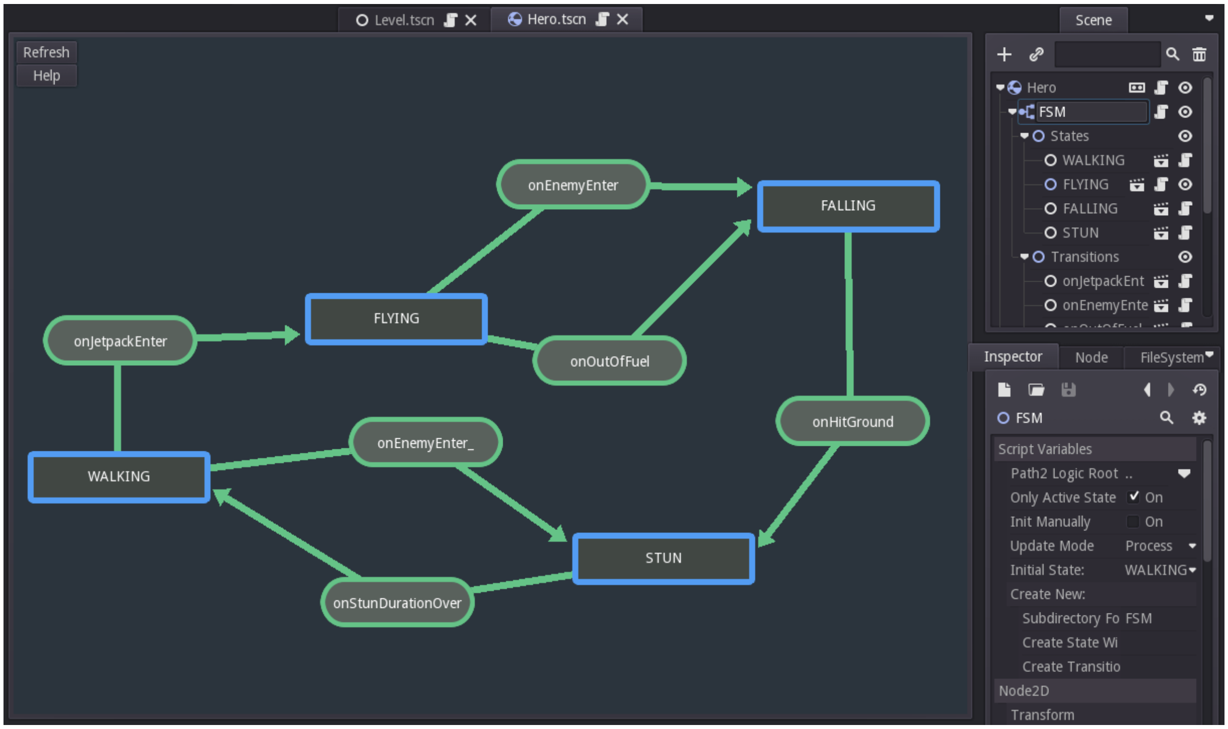
The goal is to offer a full package to create games and a continuous user experience. You can still work with external programs as long as there is an import plugin available in Godot for it. Or you can create one, like the Tiled Map Importer [https://github.com/vnen/godot-tiled-importer].

That is also partly why Godot offers its own programming language GDScript along with C#. GDScript is designed for the needs of game developers and game designers, and is tightly integrated in the engine and the editor.

GDScript lets you write code using an indentation-based syntax, yet it detects types and offers a static language's quality of auto-completion. It is also optimized for gameplay code with built-in types like Vectors and Colors.

Note that with GDExtension, you can write high-performance code using compiled languages like C, C++, Rust, D, Haxe, or Swift without recompiling the engine.

Note that the 3D workspace doesn't feature as many tools as the 2D workspace. You'll need external programs or add-ons to edit terrains, animate complex characters, and so on. Godot provides a complete API to extend the editor's functionality using game code. See The Godot editor is a Godot game below.


Level.tscn
Hero.tscn
Scene
Refresh
Help
onEnemyEnter
FALLING
FLYING
onJetpackEnter
onOutOfFuel
onHitGround
onEnemyEnter_
WALKING
STUN
onStunDurationOver
Hero
FSM
States
WALKING
FLYING
FALLING
STUN
Transitions
onJetpackEnt
onEnemyEnte
Inspector
Node
FileSystem
FSM
Script Variables
Create New:
Subdirectory Fo FSM
Create State Wi
Create Transitio
Node2D
Transform

Path2 Logic Root ..
Only Active State ✓ On
Init Manually ☐ On
Update Mode Process
Initial State: WALKING

A State Machine editor plugin in Godot 2 by kubecz3k. It lets you manage states and transitions visually.

Open source

Godot offers a fully open source codebase under the **MIT license**. This means all the technologies that ship with it have to be Free (as in freedom) as well. For the most part, they're developed from the ground up by contributors.

Anyone can plug in proprietary tools for the needs of their projects — they just won't ship with the engine. This may include Google

AdMob, or FMOD. Any of these can come as third-party plugins instead.

On the other hand, an open codebase means you can **learn from and extend the engine** to your heart's content. You can also debug games easily, as Godot will print errors with a stack trace, even if they come from the engine itself.

Note

This **does not affect the work you do with Godot** in any way: there's no strings attached to the engine or anything you make with it.

Community-driven

Godot is made by its community, for the community, and for all game creators out there. It's the needs of the users and open discussions that drive the core updates. New features from the core developers often focus on what will benefit the most users first.

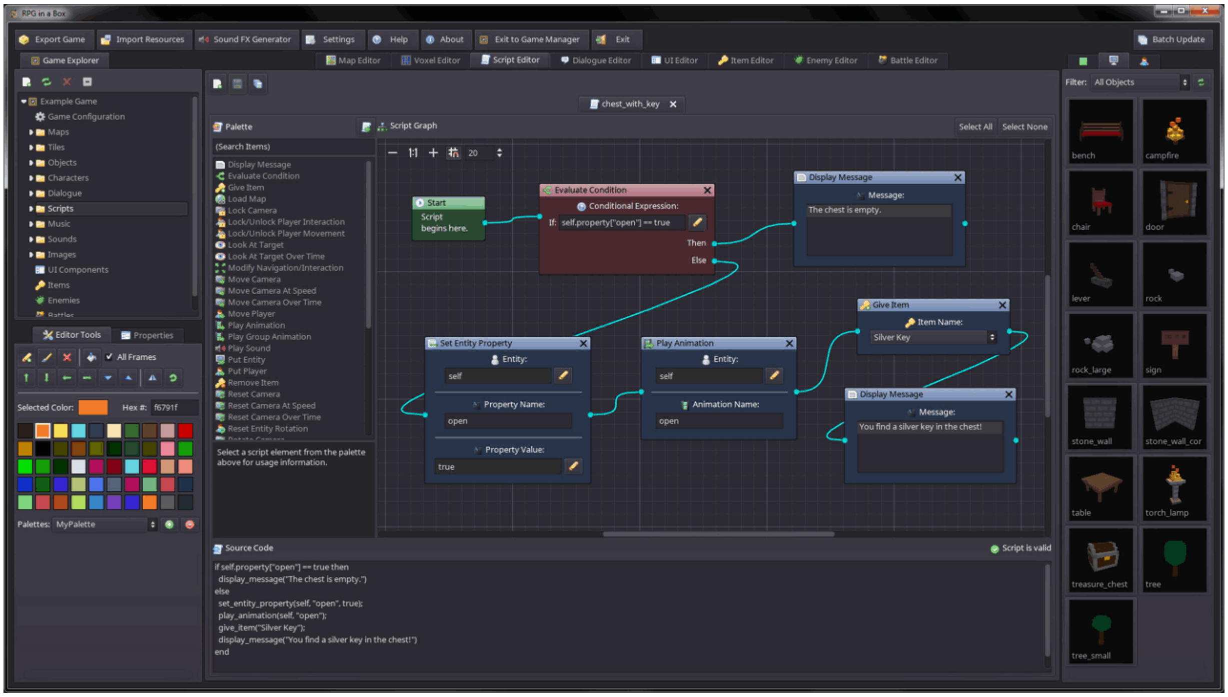
That said, although a handful of core developers work on it full-time, the project has thousands of contributors at the time of writing. Benevolent programmers work on features they may need themselves, so you'll see improvements in all corners of the engine at the same time in every major release.

The Godot editor is a Godot game

The Godot editor runs on the game engine. It uses the engine's own UI system, it can hot-reload code and scenes when you test your projects, or run game code in the editor. This means you can **use**

the same code and scenes for your games, or **build plugins and extend the editor.**

This leads to a reliable and flexible UI system, as it powers the editor itself. With the `@tool` annotation, you can run any game code in the editor.



RPG in a Box
Export Game
Import Resources
Sound FX Generator
Settings
Help
About
Exit to Game Manager
Exit
Batch Update
Game Explorer
Map Editor
Voxel Editor
Script Editor
Dialogue Editor
UI Editor
Item Editor
Enemy Editor
Battle Editor
chest_with_key
Palette
Script Graph
Select All
Select None
Start
Evaluate Condition
Display Message
Give Item
Set Entity Property
Play Animation
Source Code
Script is valid

```
if self.property["open"] == true then
 display_message("The chest is empty.")
else
 set_entity_property(self, "open", true);
 play_animation(self, "open");
 give_item("Silver Key");
 display_message("You find a silver key in the chest!")
end
```

RPG in a Box is a voxel RPG editor made with Godot 2. It uses Godot's UI tools for its node-based programming system and for the rest of the interface.

Put the `@tool` annotation at the top of any GDScript file and it will run in the editor. This lets you import and export plugins, create plugins like custom level editors, or create scripts with the same nodes and API you use in your projects.

Note

The editor is fully written in C++ and is statically compiled into the binary. This means you can't import it as a typical project that would have a `project.godot` file.

Separate 2D and 3D engines

Godot offers dedicated 2D and 3D rendering engines. As a result, **the base unit for 2D scenes is pixels.** Even though the engines are separate, you can render 2D in 3D, 3D in 2D, and overlay 2D sprites and interfaces over your 3D world.

Step by step

This series builds upon the Introduction to Godot and will get you started with the editor and the engine. You will learn more about nodes and scenes, code your first classes with GDScript, use signals to make nodes communicate with one another, and more.

The following lessons are here to prepare you for Your first 2D game, a step-by-step tutorial where you will code a game from scratch. By the end of it, you will have the necessary foundations to explore more features in other sections. We also included links to pages that cover a given topic in-depth where appropriate.

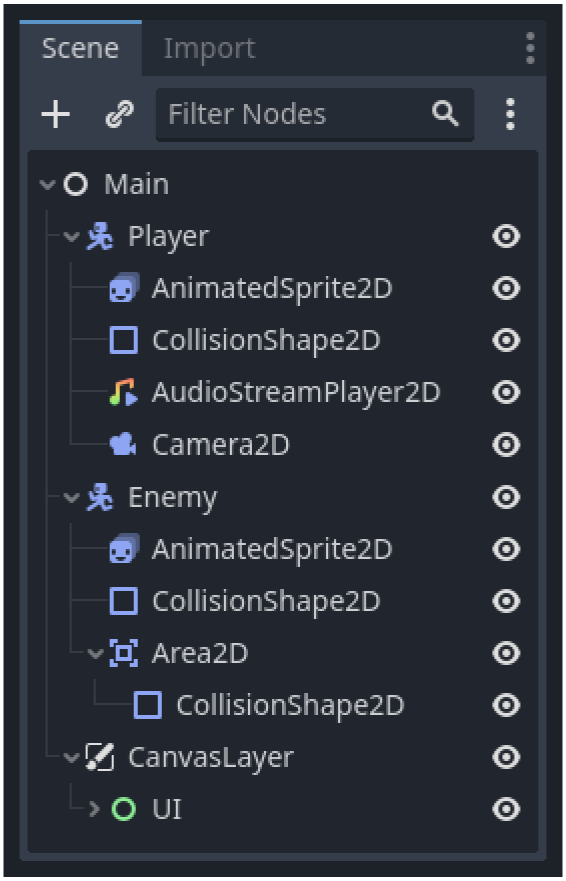
- Nodes and Scenes
- Creating instances
- Scripting languages
- Creating your first script
- Listening to player input
- Using signals

Nodes and Scenes

In Overview of Godot's key concepts, we saw that a Godot game is a tree of scenes and that each scene is a tree of nodes. In this lesson, we explain a bit more about them. You will also create your first scene.

Nodes

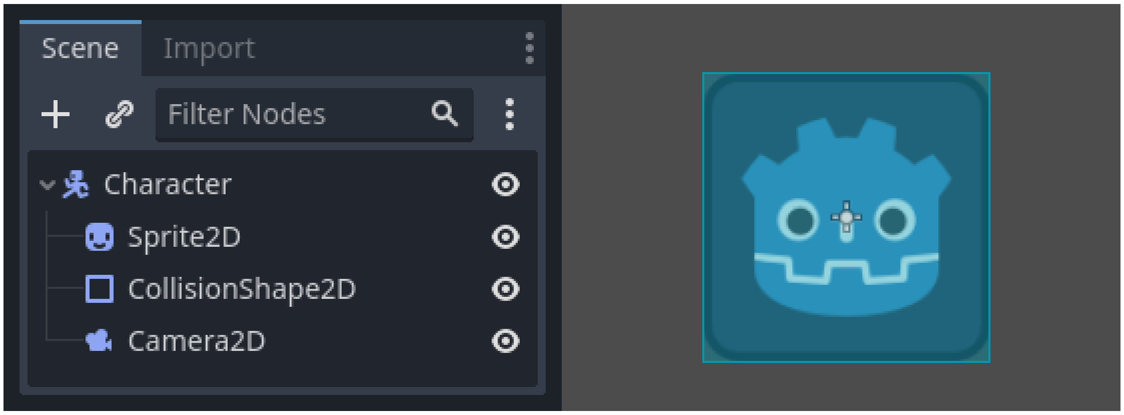
Nodes are the fundamental building blocks of your game. They are like the ingredients in a recipe. There are dozens of kinds that can display an image, play a sound, represent a camera, and much more.


Scene
Import
Filter Nodes
Main
Player
AnimatedSprite2D
CollisionShape2D
AudioStreamPlayer2D
Camera2D
Enemy
AnimatedSprite2D
CollisionShape2D
Area2D
CollisionShape2D
CanvasLayer
UI

All nodes have the following characteristics:

- A name.
- Editable properties.
- They receive callbacks to update every frame.
- You can extend them with new properties and functions.
- You can add them to another node as a child.

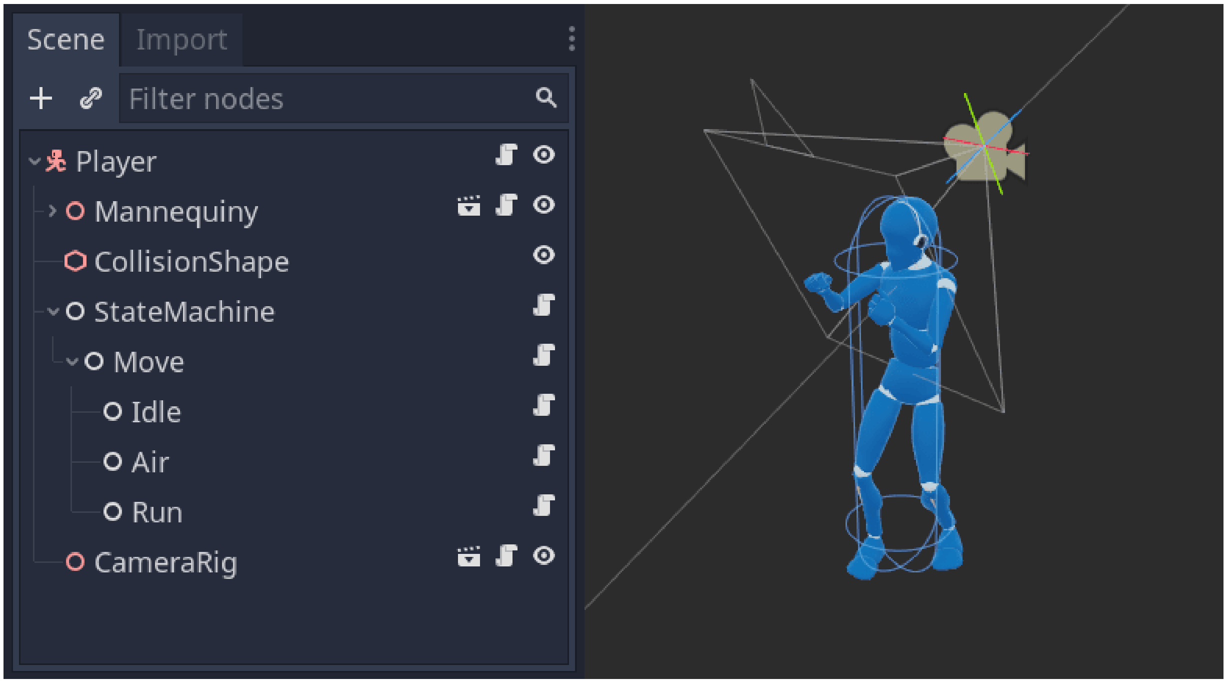
The last characteristic is important. **Together, nodes form a tree**, which is a powerful feature to organize projects. Since different nodes have different functions, combining them produces more complex behavior. As we saw before, you can build a playable character the camera follows using a CharacterBody2D node, a Sprite2D node, a Camera2D node, and a CollisionShape2D node.



Scenes

When you organize nodes in a tree, like our character, we call this construct a scene. Once saved, scenes work like new node types in the editor, where you can add them as a child of an existing node. In that case, the instance of the scene appears as a single node with its internals hidden.

Scenes allow you to structure your game's code however you want. You can **compose nodes** to create custom and complex node types, like a game character that runs and jumps, a life bar, a chest with which you can interact, and more.


Scene
Import
Filter nodes
Player
Mannequiny
CollisionShape
StateMachine
Move
Idle
Air
Run
CameraRig

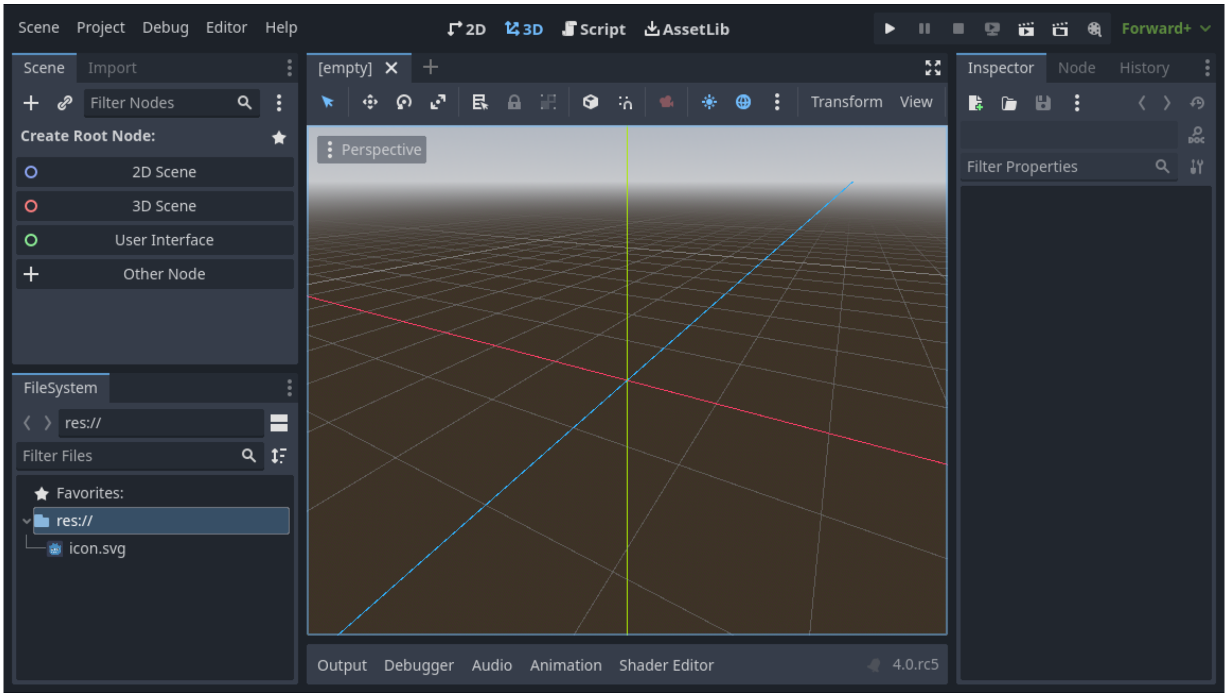
The Godot editor essentially is a **scene editor**. It has plenty of tools for editing 2D and 3D scenes, as well as user interfaces. A Godot project can contain as many of these scenes as you need. The engine only requires one as your application's **main scene**. This is the scene Godot will first load when you or a player runs the game.

On top of acting like nodes, scenes have the following characteristics:

1. They always have one root node, like the "Player" in our example.
2. You can save them to your local drive and load them later.
3. You can create as many instances of a scene as you'd like. You could have five or ten characters in your game, created from your Character scene.

Creating your first scene

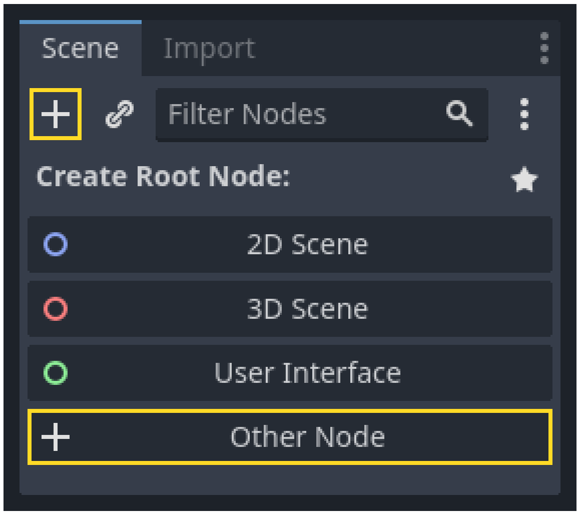
Let's create our first scene with a single node. To do so, you will need to create a new project first. After opening the project, you should see an empty editor.


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Forward+
Scene
Import
Filter Nodes
Create Root Node:
2D Scene
3D Scene
User Interface
Other Node
FileSystem
res://
Filter Files
Favorites:
res://
icon.svg
[empty]
Transform
View
Perspective
Output
Debugger
Audio
Animation
Shader Editor
4.0.rc5
Inspector
Node
History
Filter Properties

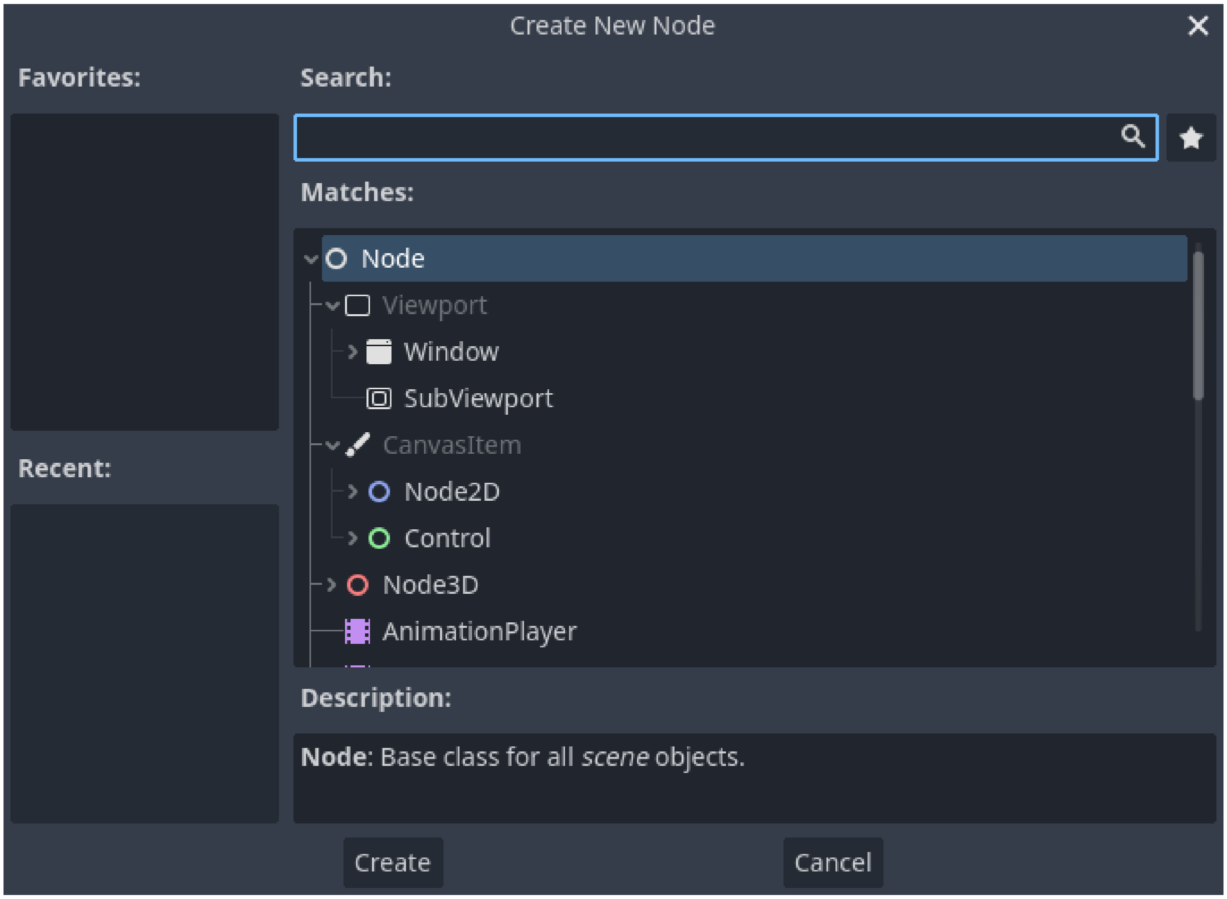
In an empty scene, the Scene dock on the left shows several options to add a root node quickly. "2D Scene" adds a Node2D node, "3D Scene" adds a Node3D node, and "User Interface" adds a Control node. These presets are here for convenience; they are not mandatory. "Other Node" lets you select any node to be the root node. In an empty scene, "Other Node" is equivalent to pressing the "Add Child Node" button at the top-left of the Scene dock, which usually adds a new node as a child of the currently selected node.

We're going to add a single Label node to our scene. Its function is to draw text on the screen.

Press the "Add Child Node" button or "Other Node" to create a root node.

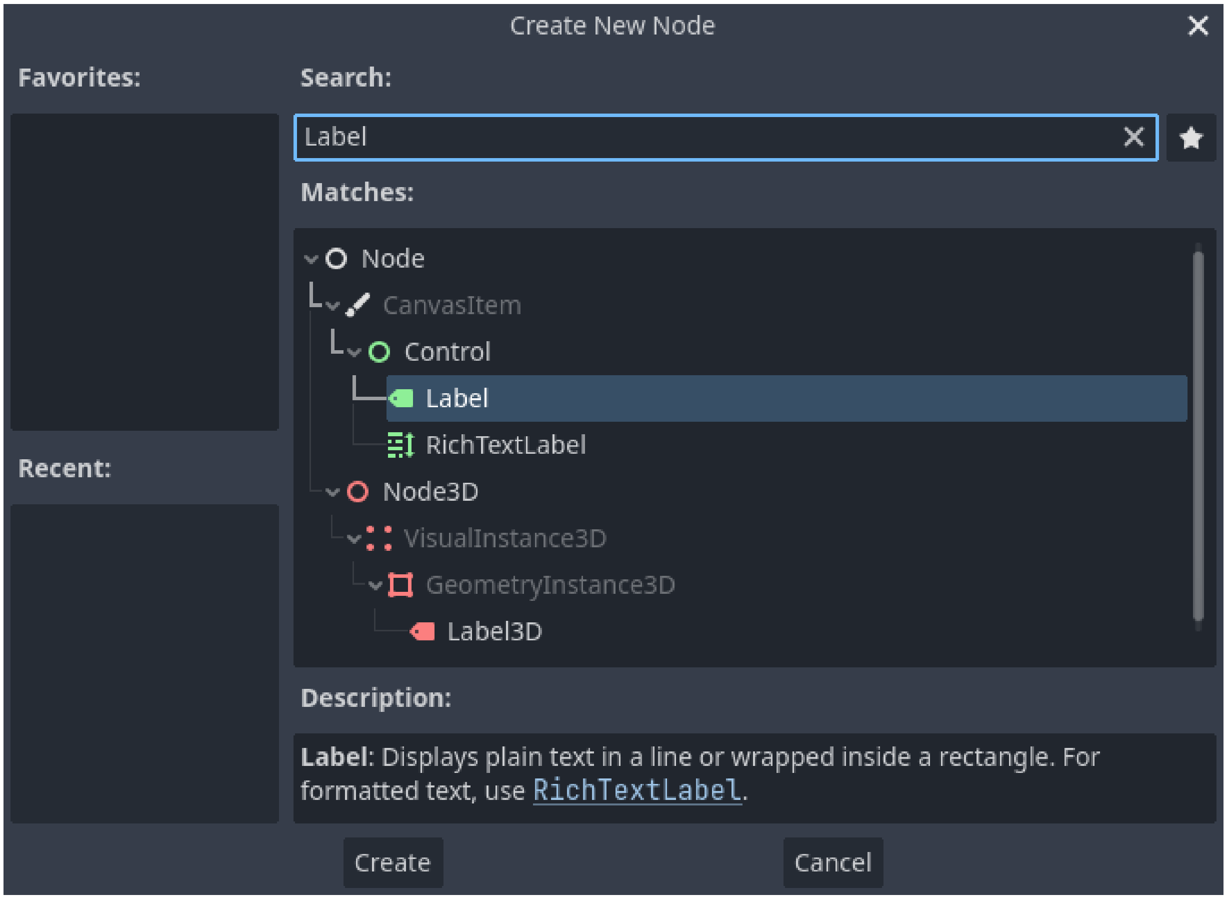

Scene
Import
Filter Nodes
Create Root Node:
2D Scene
3D Scene
User Interface
Other Node

The Create Node dialog opens, showing the long list of available nodes.


Create New Node
Favorites:
Search:
Matches:
Node
Viewport
Window
SubViewport
CanvasItem
Node2D
Control
Node3D
AnimationPlayer
Recent:
Description:
Create
Cancel

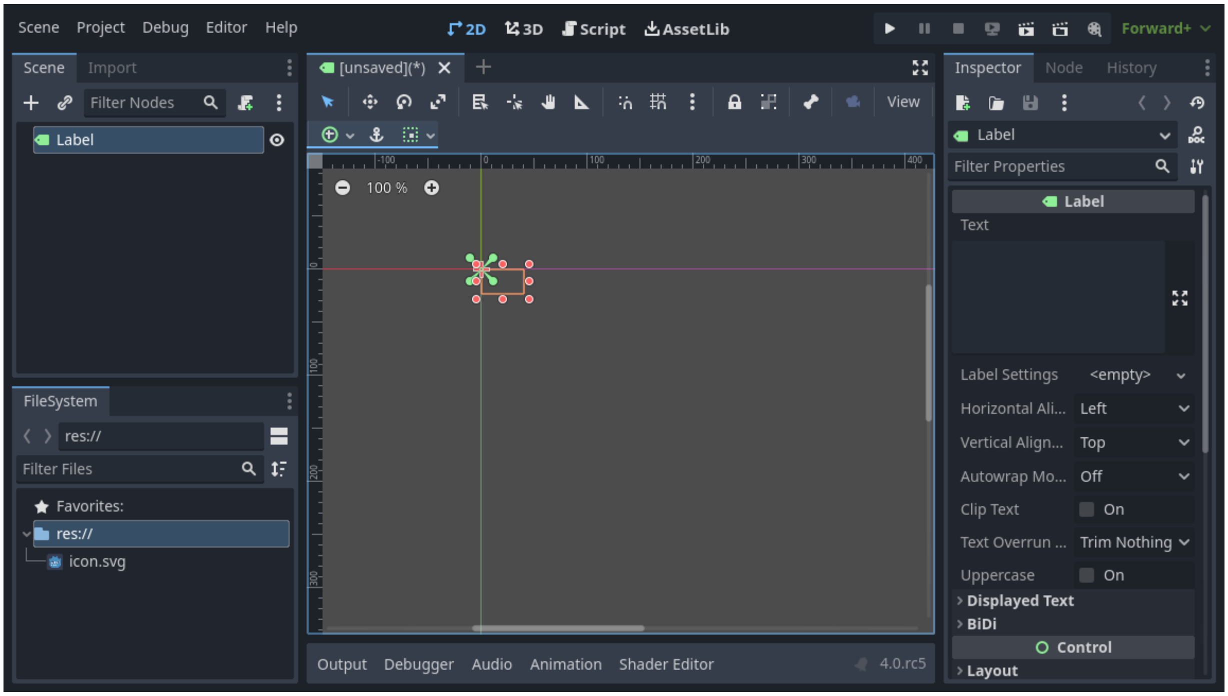
Node: Base class for all *scene* objects.

Select the Label node. You can type its name to filter down the list.


Create New Node
Favorites:
Search:
Label
Matches:
Node
CanvasItem
Control
Label
RichTextLabel
Node3D
VisualInstance3D
GeometryInstance3D
Label3D
Description:
Recent:
Create
Cancel

Label: Displays plain text in a line or wrapped inside a rectangle. For formatted text, use `RichTextLabel`.

Click on the Label node to select it and click the Create button at the bottom of the window.

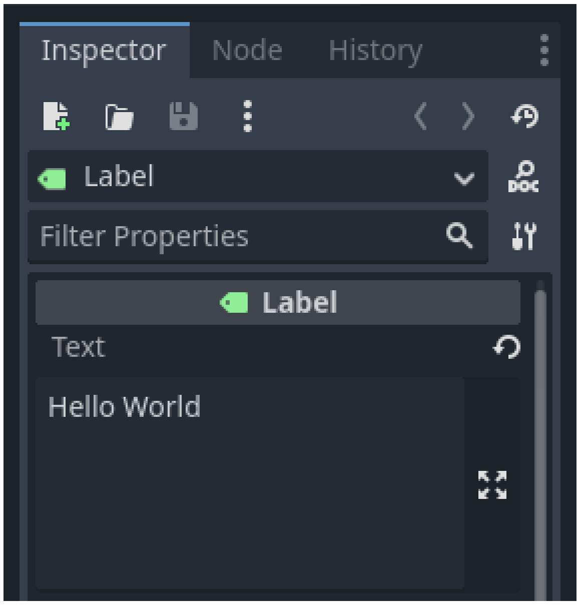

Scene Project Debug Editor Help
2D 3D Script AssetLib
Forward+
Scene Import
Filter Nodes
Label
[unsaved](*)
View
100 %
FileSystem
res://
Filter Files
Favorites:
res://
icon.svg
Output Debugger Audio Animation Shader Editor
4.0.rc5
Inspector Node History
Label
Filter Properties
Label
Text
Label Settings <empty>
Horizontal Ali... Left
Vertical Align... Top
Autowrap Mo... Off
Clip Text On
Text Overrun ... Trim Nothing
Uppercase On
Displayed Text
BiDi
Control
Layout

A lot happens when you add a scene's first node. The scene changes to the 2D workspace because Label is a 2D node type. The Label appears, selected, in the top-left corner of the viewport. The node appears in the Scene dock on the left, and the node's properties appear in the Inspector dock on the right.

Changing a node's properties

The next step is to change the Label's "Text" property. Let's change it to "Hello World".

Head to the Inspector dock on the right of the viewport. Click inside the field below the Text property and type "Hello World".


Inspector
Node
History
Label
Filter Properties
Label

Text

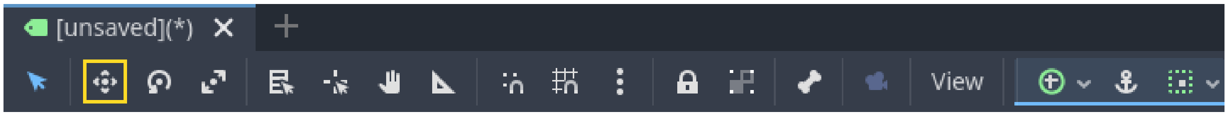
Hello World

You will see the text draw in the viewport as you type.

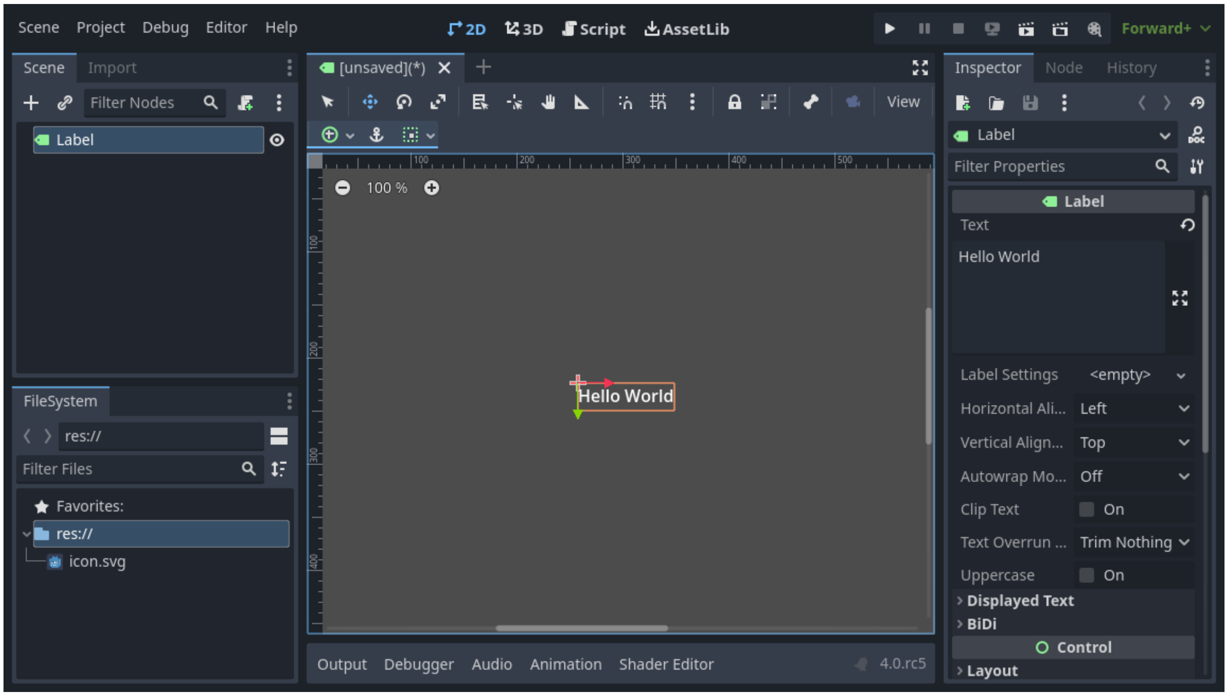
See also

You can edit any property listed in the Inspector as we did with the Text. For a complete reference of the Inspector dock, see [The Inspector](#).

You can move your Label node in the viewport by selecting the move tool in the toolbar.


[unsaved](*)
View

With the Label selected, click and drag anywhere in the viewport to move it to the center of the view delimited by the rectangle.


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Forward+
Scene
Import
Filter Nodes
Label
[unsaved](*)
View
100 %
Hello World
FileSystem
res://
Filter Files
Favorites:
res://
icon.svg
Output
Debugger
Audio
Animation
Shader Editor
4.0.rc5
Inspector
Node
History
Label
Filter Properties
Label
Displayed Text
BiDi
Control
Layout

Text: Hello World

Label Settings: <empty>

Horizontal Ali...: Left

Vertical Align...: Top

Autowrap Mo...: Off

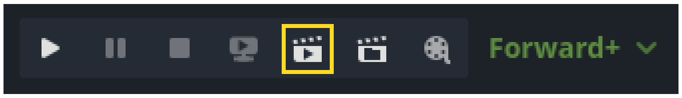
Clip Text: ☐ On

Text Overrun ...: Trim Nothing

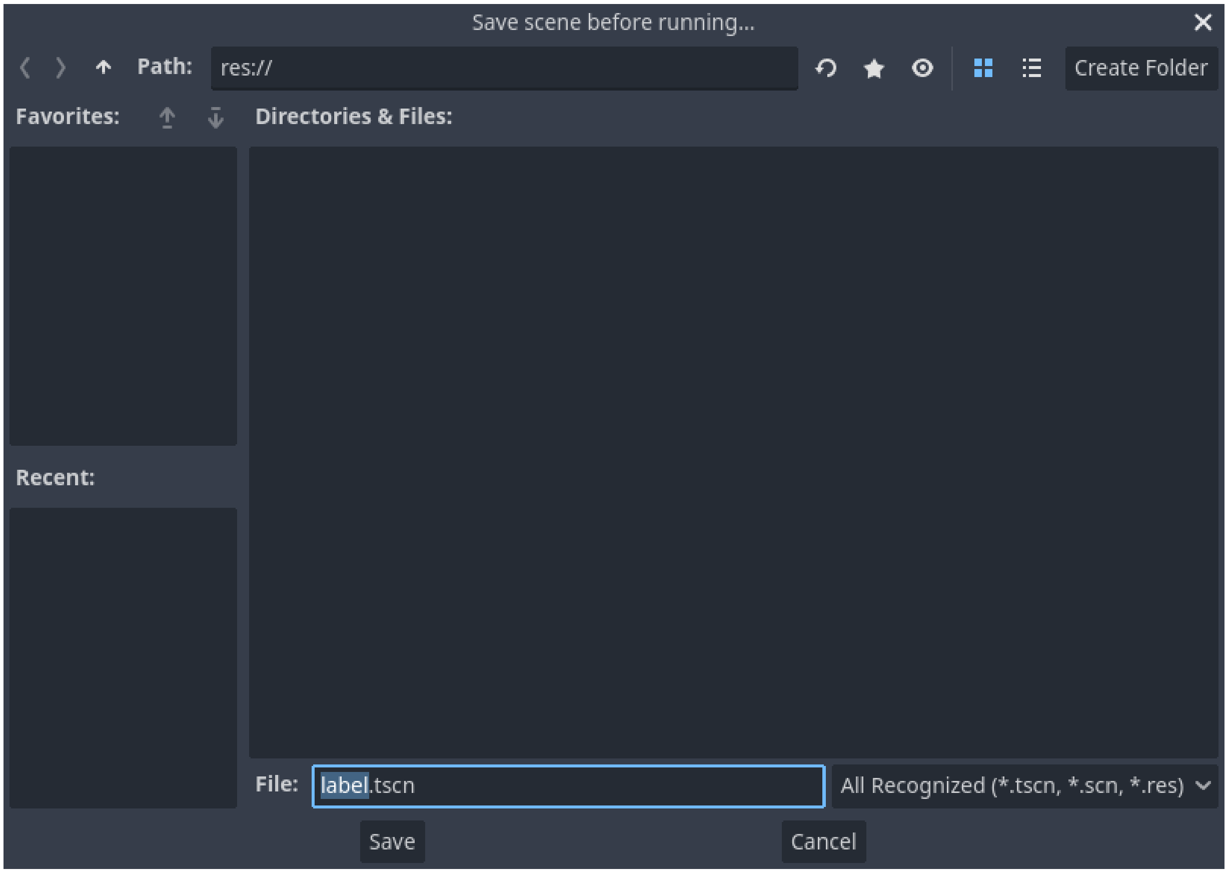
Uppercase: ☐ On

Running the scene

Everything's ready to run the scene! Press the Play Scene button in the top-right of the screen or press `F6` (`Cmd + R` on macOS).



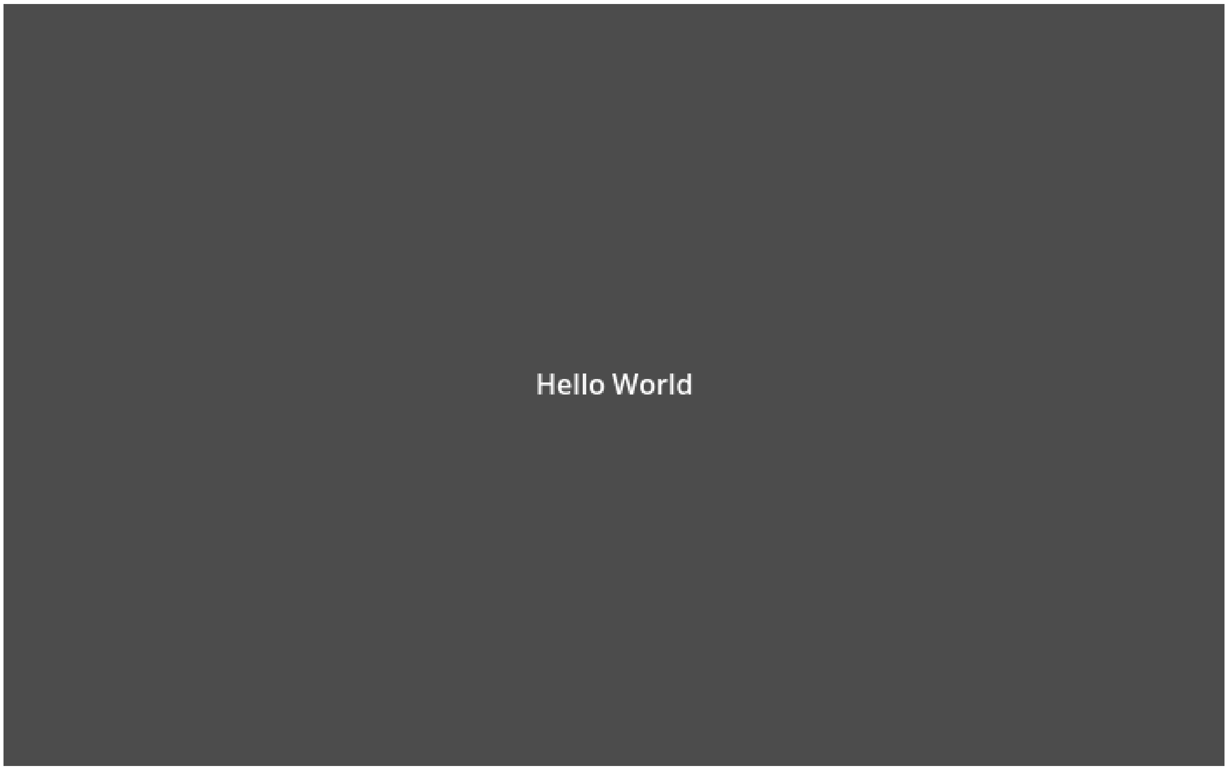
A popup invites you to save the scene, which is required to run it. Click the Save button in the file browser to save it as `label.tscn`.


Save scene before running...
Path:
res://
Create Folder
Favorites:
Directories & Files:
Recent:
File:
label.tscn
All Recognized (*.tscn, *.scn, *.res)
Save
Cancel

Note

The Save Scene As dialog, like other file dialogs in the editor, only allows you to save files inside the project. The `res://` path at the top of the window represents the project's root directory and stands for "resource path". For more information about file paths in Godot, see File system.

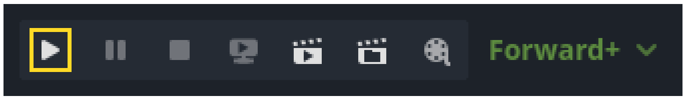
The application should open in a new window and display the text "Hello World".



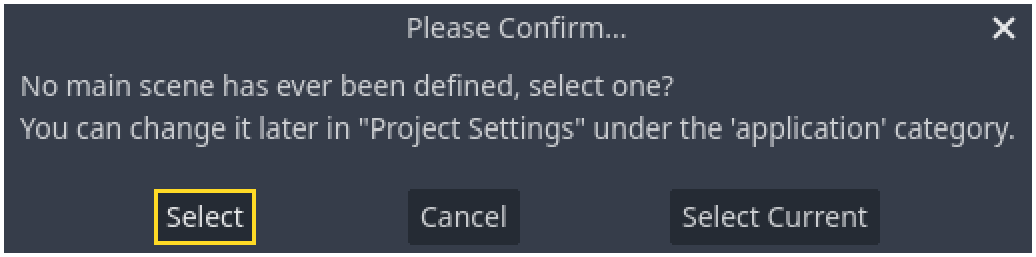
Close the window or press `F8` (`Cmd + .` on macOS) to quit the running scene.

Setting the main scene

To run our test scene, we used the Run Current Scene button. Another button next to it allows you to set and run the project's main scene. You can press `F5` (`Cmd + B` on macOS) to do so.

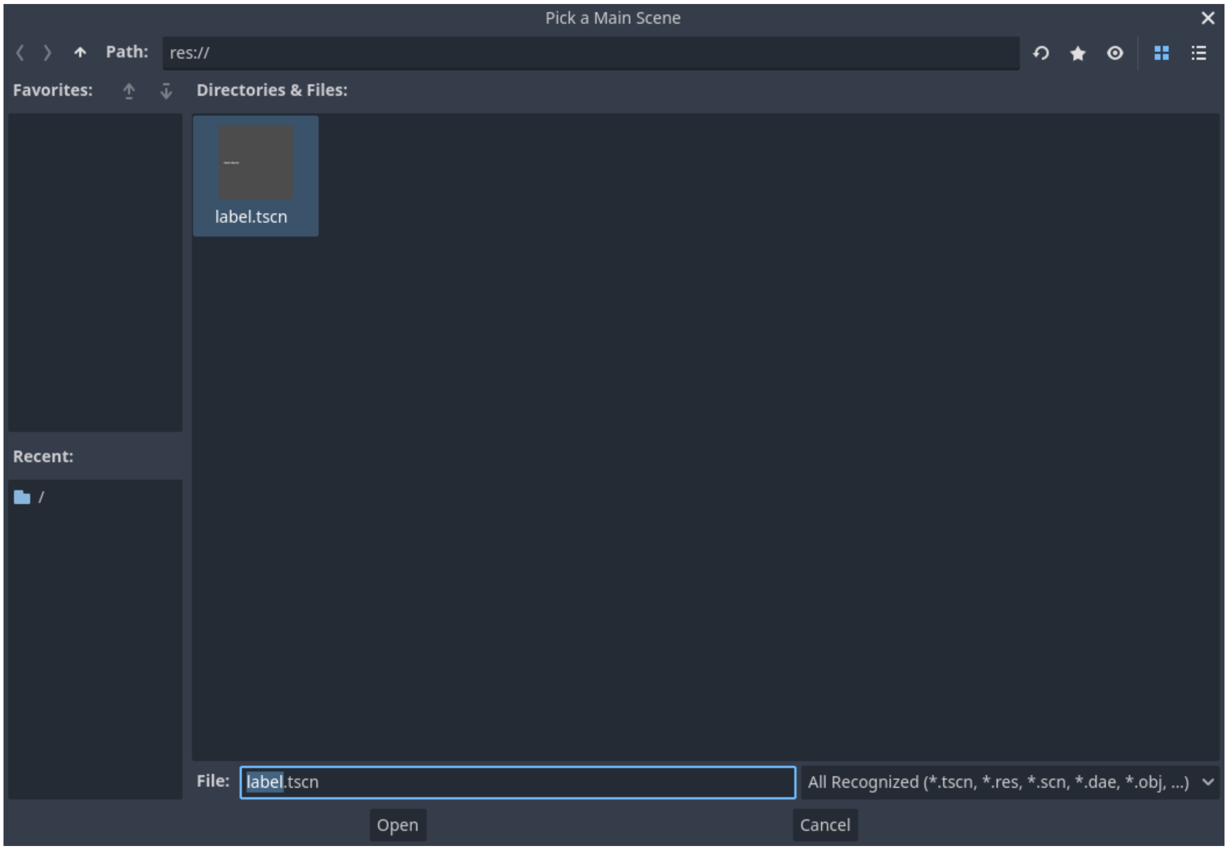


A popup window appears and invites you to select the main scene.


Please Confirm...
Select
Cancel
Select Current

No main scene has ever been defined, select one?
You can change it later in "Project Settings" under the 'application' category.

Click the Select button, and in the file dialog that appears, double click on `label.tscn`.


Pick a Main Scene
Path:
res://
Favorites:
Directories & Files:
label.tscn
Recent:
/
All Recognized (*.tscn, *.res, *.scn, *.dae, *.obj, ...)
Open
Cancel

File: label.tscn

The demo should run again. Moving forward, every time you run the project, Godot will use this scene as a starting point.

Note

The editor saves the main scene's path in a project.godot file in your project's directory. While you can edit this text file directly to change project settings, you can also use the "Project -> Project

Settings" window to do so. For more information, see Project Settings.

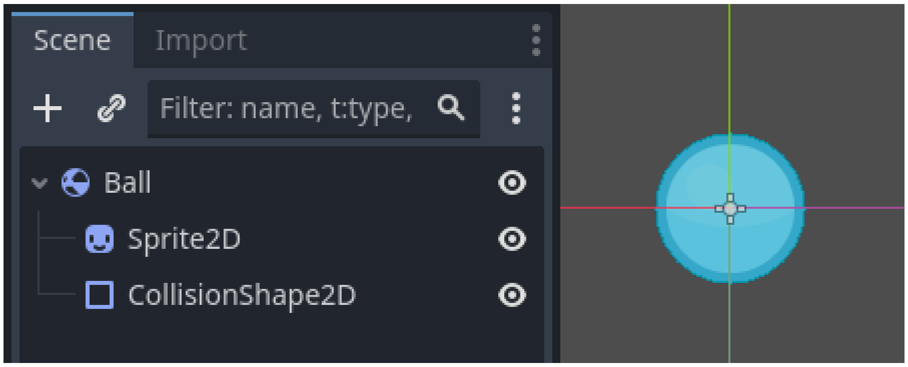
In the next part, we will discuss another key concept in games and in Godot: creating instances of a scene.

Creating instances

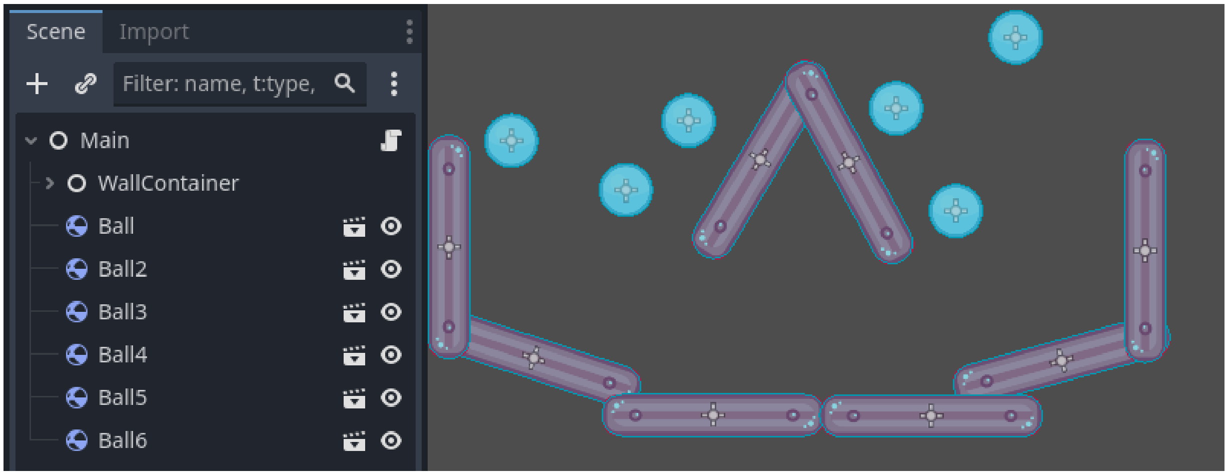
In the previous part, we saw that a scene is a collection of nodes organized in a tree structure, with a single node as its root. You can split your project into any number of scenes. This feature helps you break down and organize your game's different components.

You can create as many scenes as you'd like and save them as files with the `.tscn` extension, which stands for "text scene". The `label.tscn` file from the previous lesson was an example. We call those files "Packed Scenes" as they pack information about your scene's content.

Here's an example of a ball. It's composed of a RigidBody2D node as its root named Ball, which allows the ball to fall and bounce on walls, a Sprite2D node, and a CollisionShape2D.



Once you have saved a scene, it works as a blueprint: you can reproduce it in other scenes as many times as you'd like. Replicating an object from a template like this is called **instancing**.



As we mentioned in the previous part, instanced scenes behave like a node: the editor hides their content by default. When you instance the Ball, you only see the Ball node. Notice also how each duplicate has a unique name.

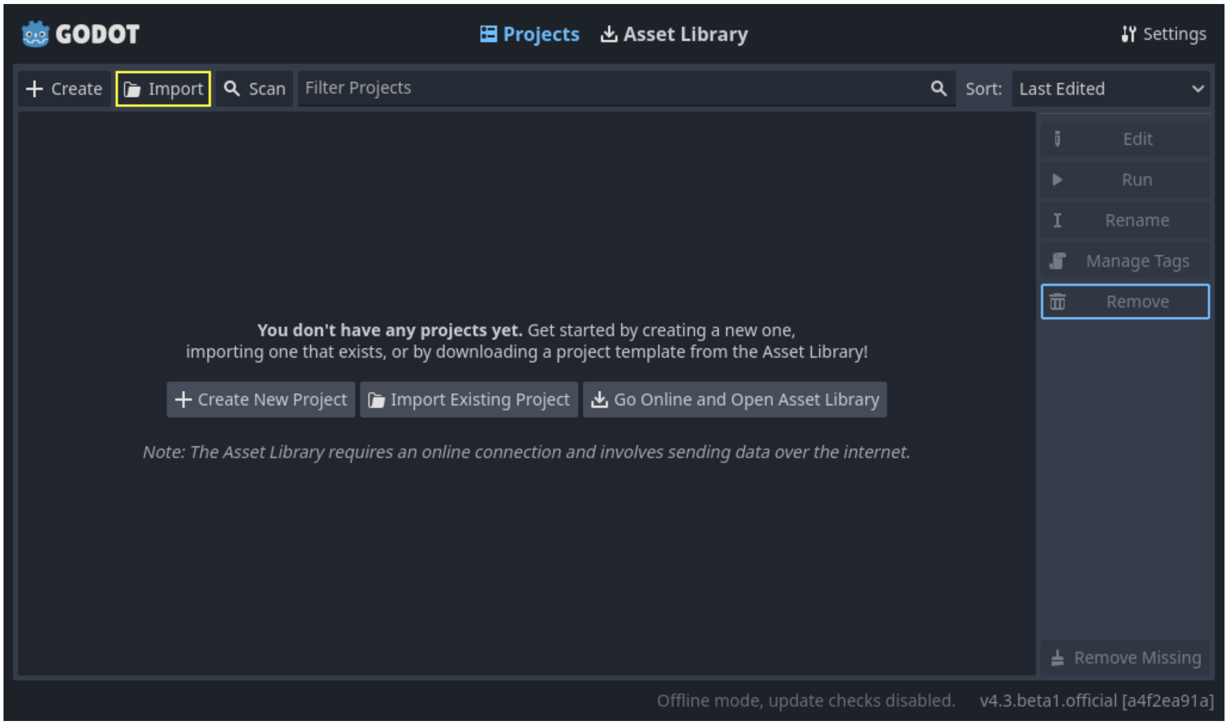
Every instance of the Ball scene starts with the same structure and properties as `ball.tscn`. However, you can modify each independently, such as changing how they bounce, how heavy they are, or any property exposed by the source scene.

In practice

Let's use instancing in practice to see how it works in Godot. We invite you to download the ball's sample project we prepared for you: instancing_starter.zip [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/instancing_starter.zip].

Extract the archive on your computer. To import it, you need the Project Manager. The Project Manager is accessed by opening Godot, or if you already have Godot opened, click on *Project -> Quit to Project List* (`Ctrl + Shift + Q`, `Ctrl + Option + Cmd + Q` on macOS)

In the Project Manager, click the *Import* button to import the project.



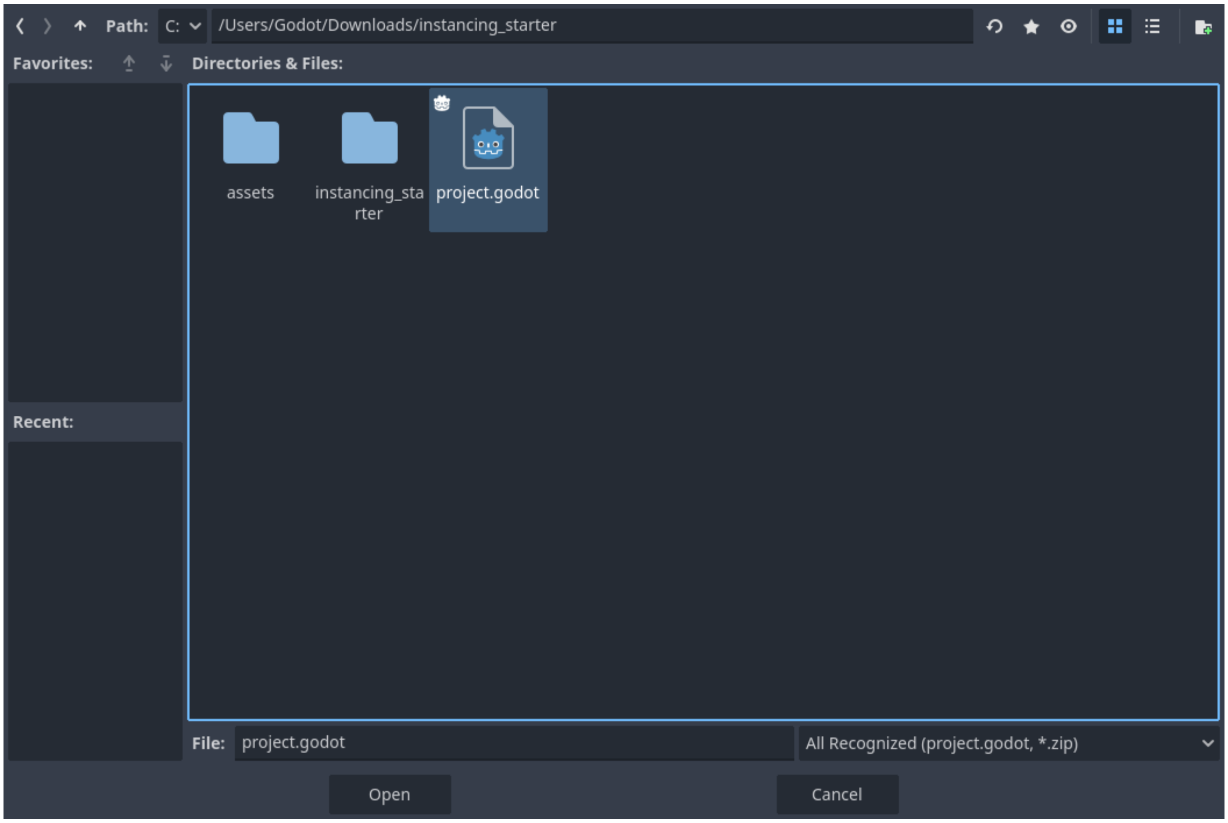
You don't have any projects yet. Get started by creating a new one, importing one that exists, or by downloading a project template from the Asset Library!

Note: The Asset Library requires an online connection and involves sending data over the internet.

Offline mode, update checks disabled. v4.3.beta1.official [a4f2ea91a]

GODOT
Projects
Asset Library
Settings
Create
Import
Scan
Filter Projects
Sort:
Last Edited
Edit
Run
Rename
Manage Tags
Remove
Create New Project
Import Existing Project
Go Online and Open Asset Library
Remove Missing

In the pop-up that appears navigate to the folder you extracted. Double-click the `project.godot` file to open it.



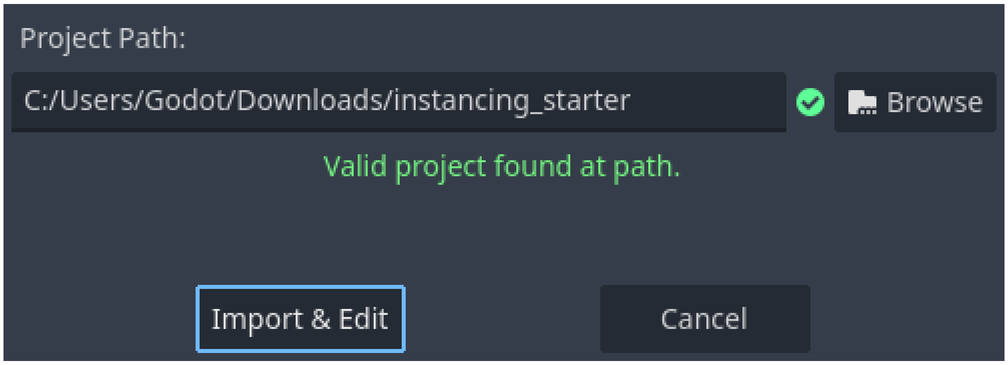
Path: C: /Users/Godot/Downloads/instancing_starter

File: project.godot

All Recognized (project.godot, *.zip)

Favorites:
Directories & Files:
assets
instancing_starter
project.godot
Recent:
Open
Cancel

Finally, click the Import & Edit button.



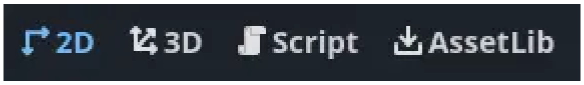
Project Path: C:/Users/Godot/Downloads/instancing_starter

Valid project found at path.

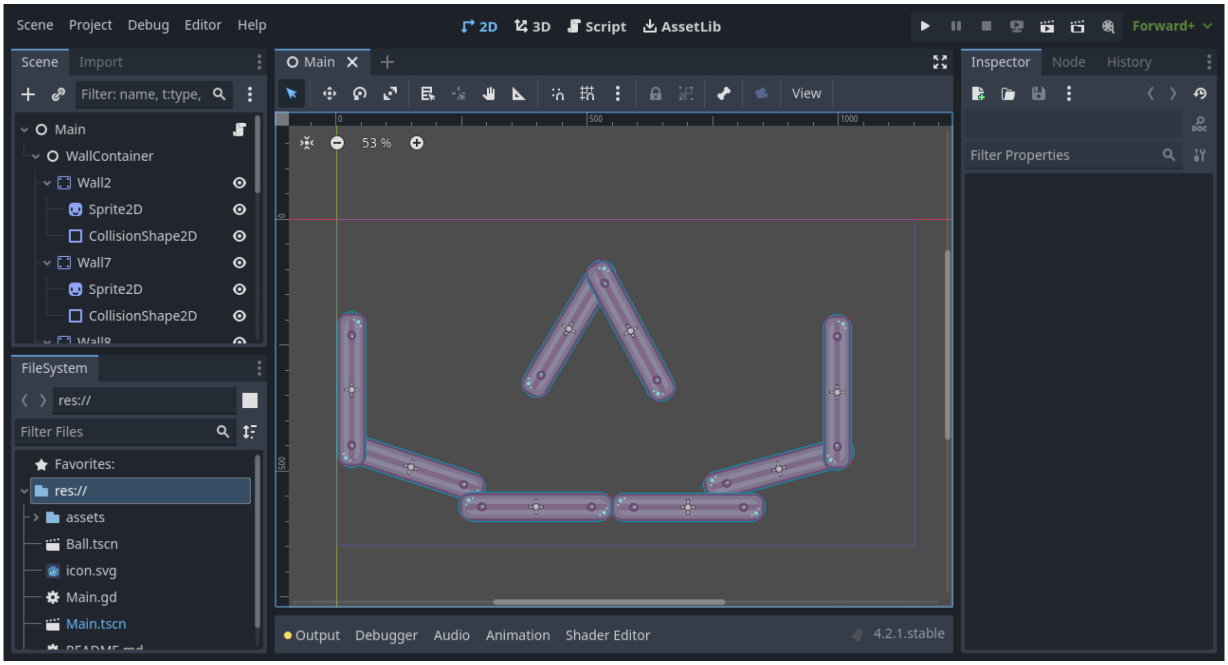
Browse
Import & Edit
Cancel

A window notifying you that the project was last opened in an older Godot version may appear, that's not an issue. Click *Ok* to open the project.

The project contains two packed scenes: `main.tscn`, containing walls against which the ball collides, and `ball.tscn`. The Main scene should open automatically. If you're seeing an empty 3D scene instead of the main scene, click the 2D button at the top of the screen.

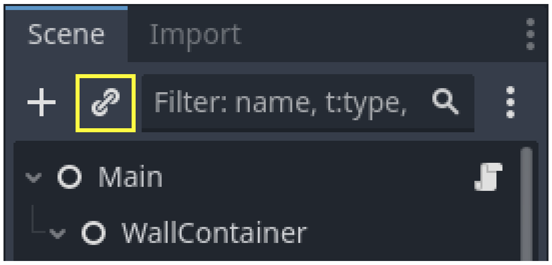


2D
3D
Script
AssetLib

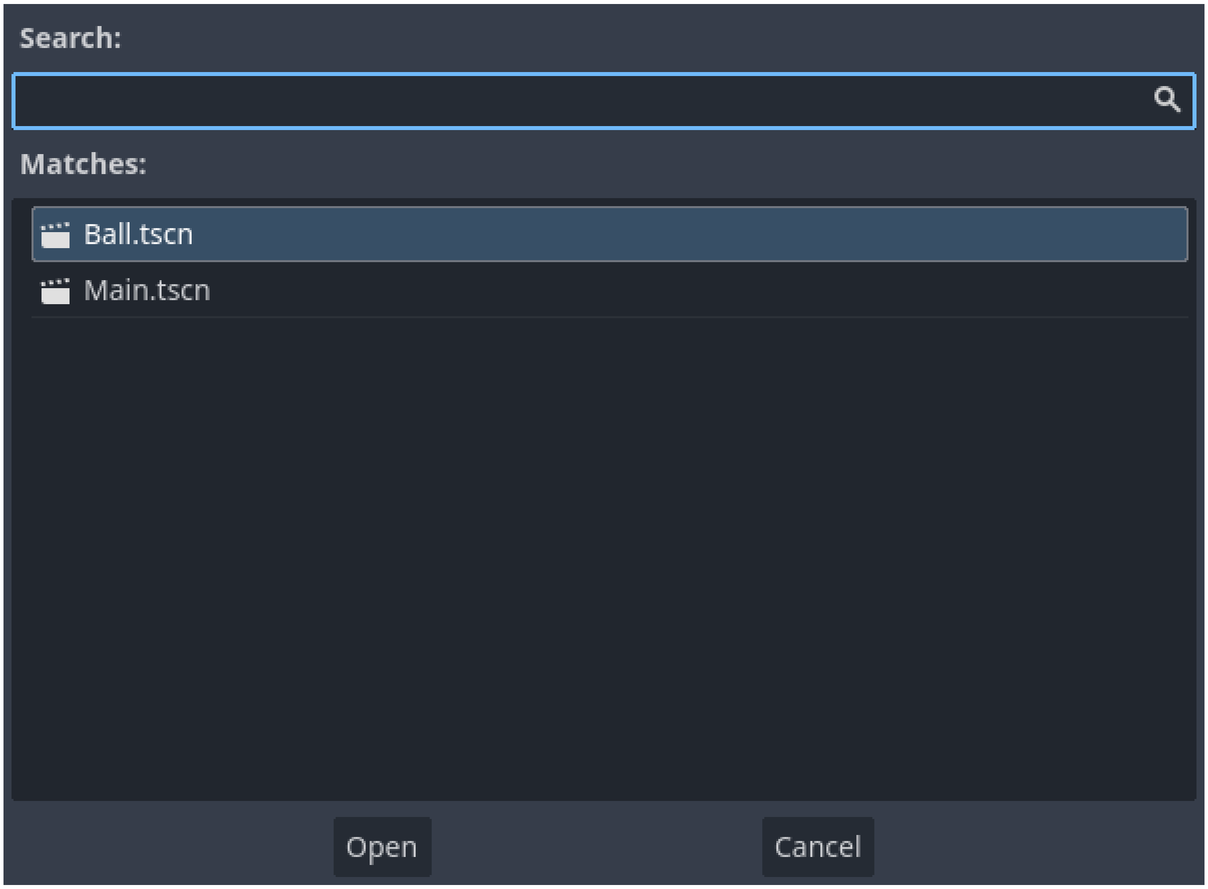


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Forward+
Scene
Import
Filter: name, t:type,
Main
WallContainer
Wall2
Sprite2D
CollisionShape2D
Wall7
Sprite2D
CollisionShape2D
FileSystem
res://
Filter Files
Favorites:
res://
assets
Ball.tscn
icon.svg
Main.gd
Main.tscn
Main
View
53 %
Inspector
Node
History
Filter Properties
Output
Debugger
Audio
Animation
Shader Editor
4.2.1.stable

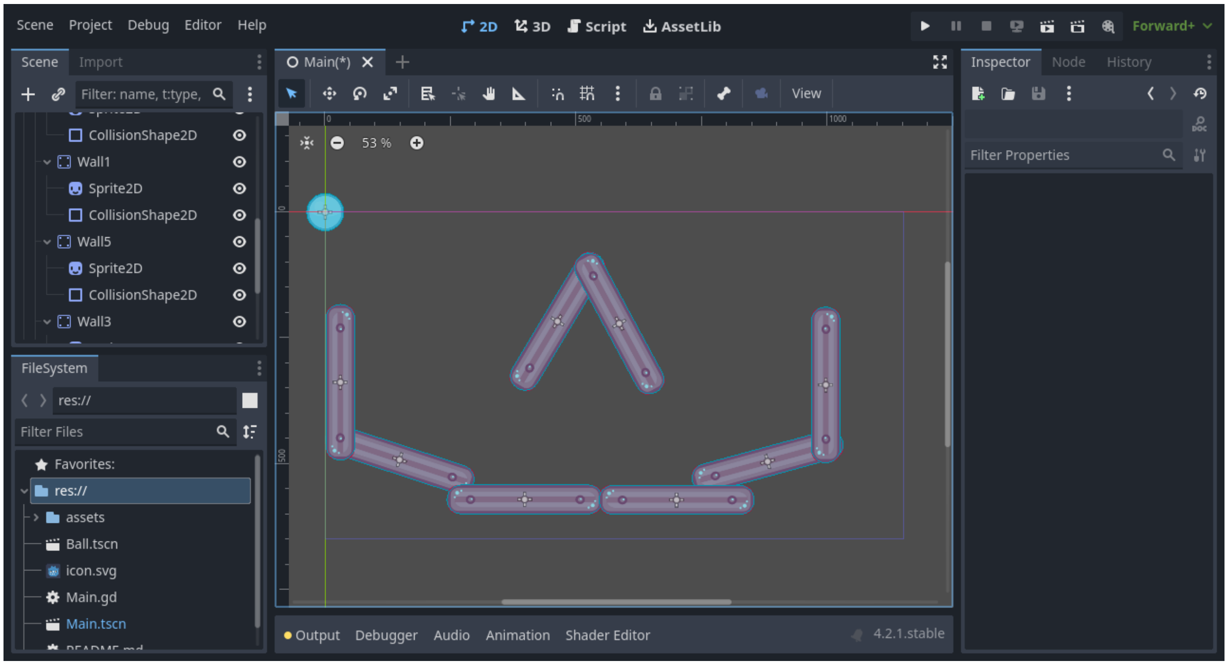
Let's add a ball as a child of the Main node. In the Scene dock, select the Main node. Then, click the link icon at the top of the scene dock. This button allows you to add an instance of a scene as a child of the currently selected node.


Scene
Import
Filter: name, t:type,
Main
WallContainer

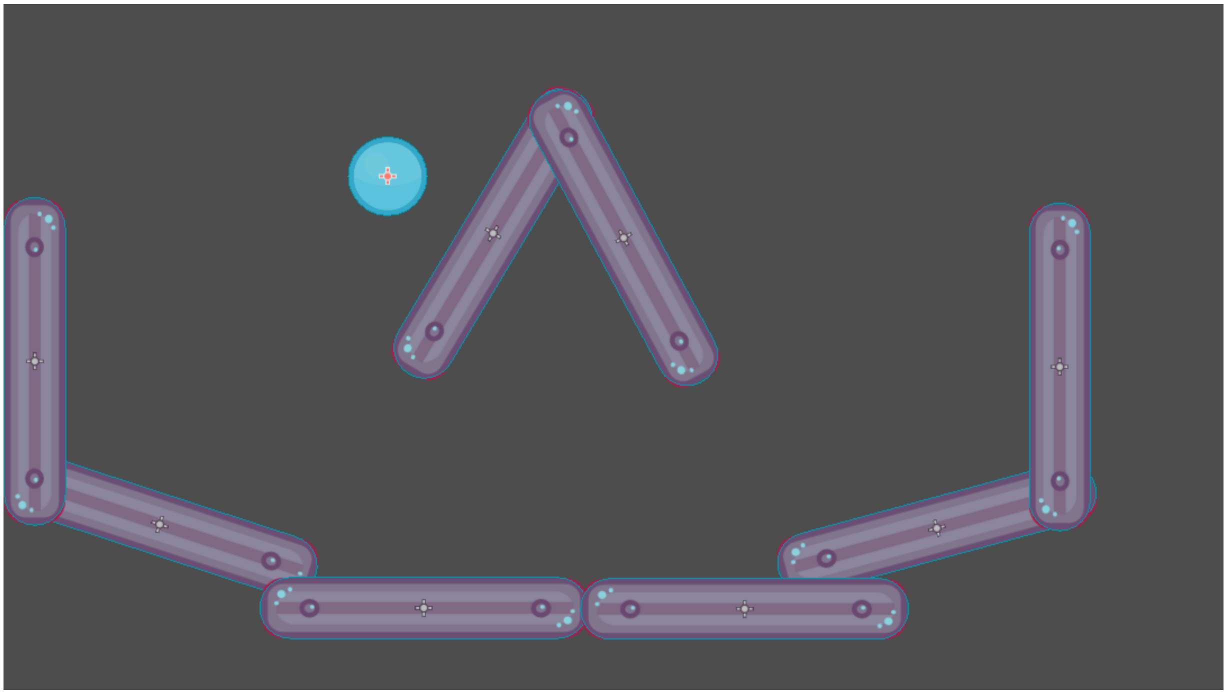
Double-click the ball scene to instance it.


Search:
Matches:
Ball.tscn
Main.tscn
Open
Cancel

The ball appears in the top-left corner of the viewport.

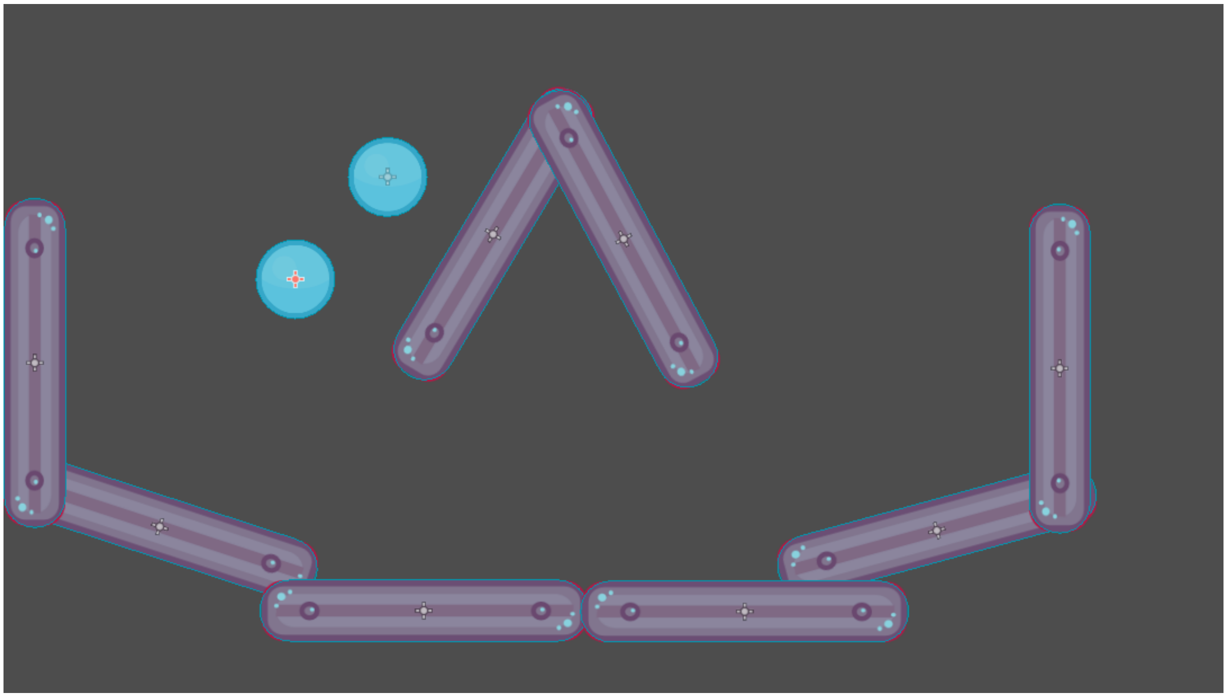

Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Forward+
Scene
Import
Filter: name, t:type,
CollisionShape2D
Wall1
Sprite2D
CollisionShape2D
Wall5
Sprite2D
CollisionShape2D
Wall3
FileSystem
res://
Filter Files
Favorites:
res://
assets
Ball.tscn
icon.svg
Main.gd
Main.tscn
README.md
Main(*)
View
53 %
Inspector
Node
History
Filter Properties
Output
Debugger
Audio
Animation
Shader Editor
4.2.1.stable

Click on it and drag it towards the center of the view.

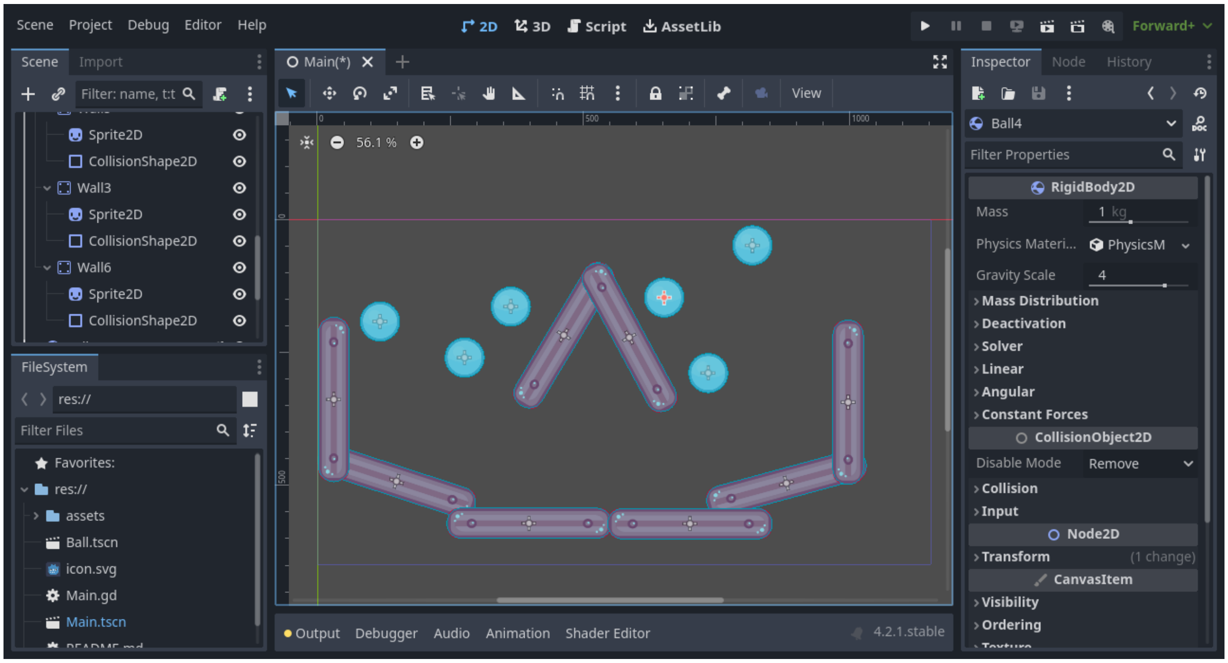


Play the game by pressing `F5` (`Cmd + B` on macOS). You should see it fall.

Now, we want to create more instances of the Ball node. With the ball still selected, press `Ctrl + D` (`Cmd + D` on macOS) to call the duplicate command. Click and drag to move the new ball to a different location.



You can repeat this process until you have several in the scene.


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Forward+
Scene
Import
Filter: name, t:t
Sprite2D
CollisionShape2D
Wall3
Sprite2D
CollisionShape2D
Wall6
Sprite2D
CollisionShape2D
FileSystem
res://
Filter Files
Favorites:
res://
assets
Ball.tscn
icon.svg
Main.gd
Main.tscn
Main(*)
View
56.1 %
0
500
1000
Output
Debugger
Audio
Animation
Shader Editor
4.2.1.stable
Inspector
Node
History
Ball4
Filter Properties
RigidBody2D
Physics Materi...
PhysicsM
Mass Distribution
Deactivation
Solver
Linear
Angular
Constant Forces
CollisionObject2D
Collision
Input
Node2D
Transform
(1 change)
CanvasItem
Visibility
Ordering

Mass 1 kg

Gravity Scale 4

Disable Mode Remove

Play the game again. You should now see every ball fall independently from one another. This is what instances do. Each is an independent reproduction of a template scene.

Editing scenes and instances

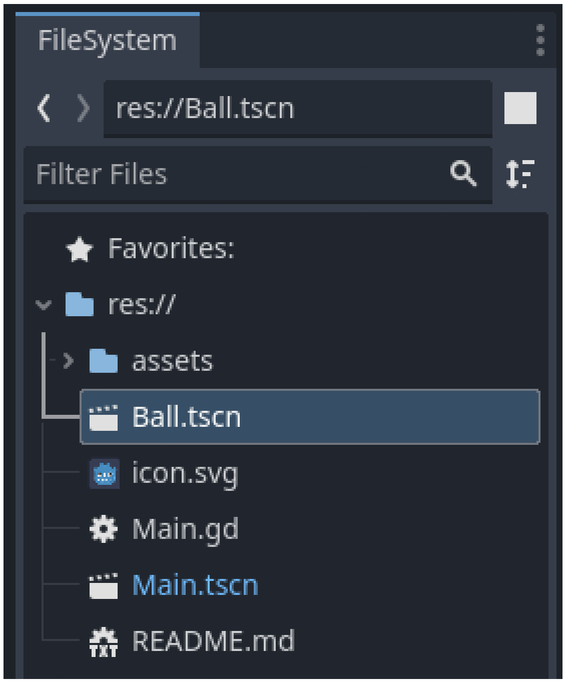
There is more to instances. With this feature, you can:

1. Change the properties of one ball without affecting the others using the Inspector.
2. Change the default properties of every Ball by opening the `ball.tscn` scene and making a change to the Ball node there. Upon saving, all instances of the Ball in the project will see their values update.

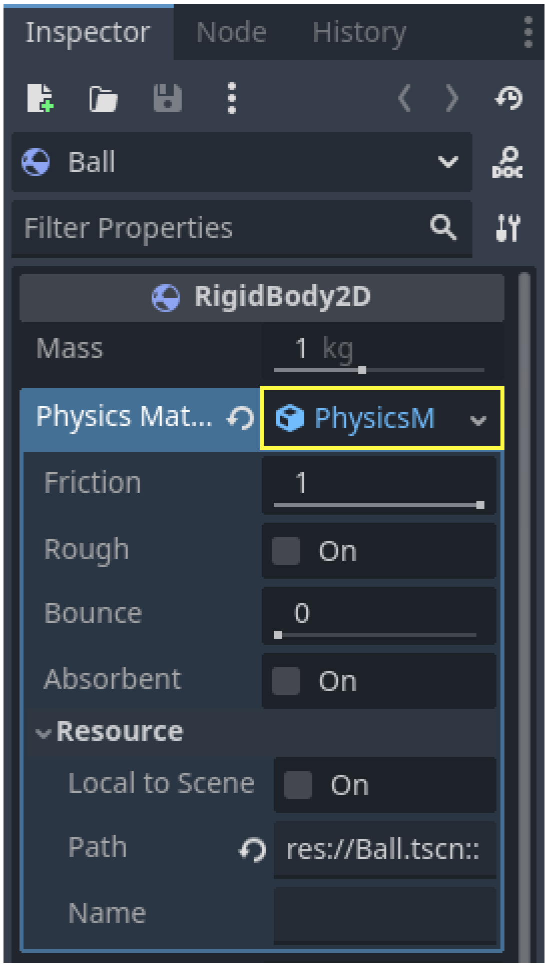
Note

Changing a property on an instance always overrides values from the corresponding packed scene.

Let's try this. Double-click `ball.tscn` in the FileSystem to open it.

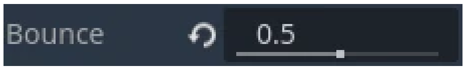

FileSystem
res://Ball.tscn
Filter Files
Favorites:
res://
assets
Ball.tscn
icon.svg
Main.gd
Main.tscn
README.md

Select the Ball node. In the Inspector on the right, click on the PhysicsMaterial property to expand it.


Inspector
Node
History
Ball
Filter Properties
RigidBody2D
Physics Mat...
PhysicsM
Resource

Mass 1 kg
Friction 1
Rough ☐ On
Bounce 0
Absorbent ☐ On
Local to Scene ☐ On
Path res://Ball.tscn::
Name

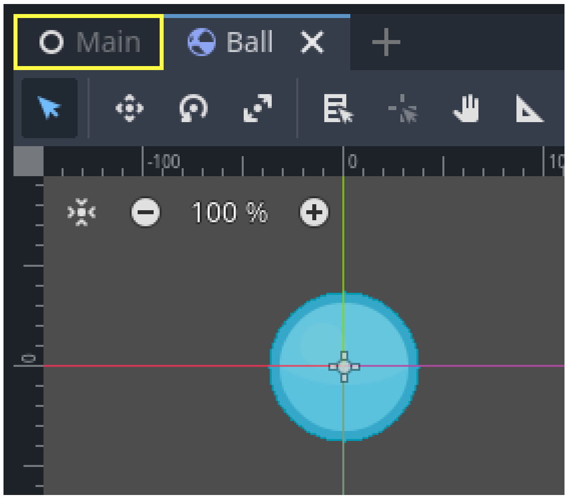
Set its Bounce property to `0.5` by clicking on the number field, typing `0.5`, and pressing `Enter`.



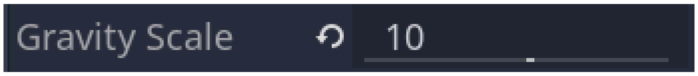
Bounce 0.5

Play the game by pressing `F5` (`Cmd + B` on macOS) and notice how all balls now bounce a lot more. As the Ball scene is a template for all instances, modifying it and saving causes all instances to update accordingly.

Let's now adjust an individual instance. Head back to the Main scene by clicking on the corresponding tab above the viewport.

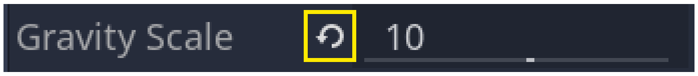


Select one of the instanced Ball nodes and, in the Inspector, set its Gravity Scale value to `10`.



Gravity Scale 10

A grey "revert" button appears next to the adjusted property.



Gravity Scale 10

This icon indicates you are overriding a value from the source packed scene. Even if you modify the property in the original scene, the value override will be preserved in the instance. Clicking the revert icon will restore the property to the value in the saved scene.

Rerun the game and notice how this ball now falls much faster than the others.

Note

You may notice you are unable to change the values of the PhysicsMaterial of the ball. This is because PhysicsMaterial is a *resource*, and needs to be made unique before you can edit it in a scene that is linking to its original scene. To make a resource unique for one instance, right-click on the **Physics Material** property in the Inspector and click **Make Unique** in the context menu.

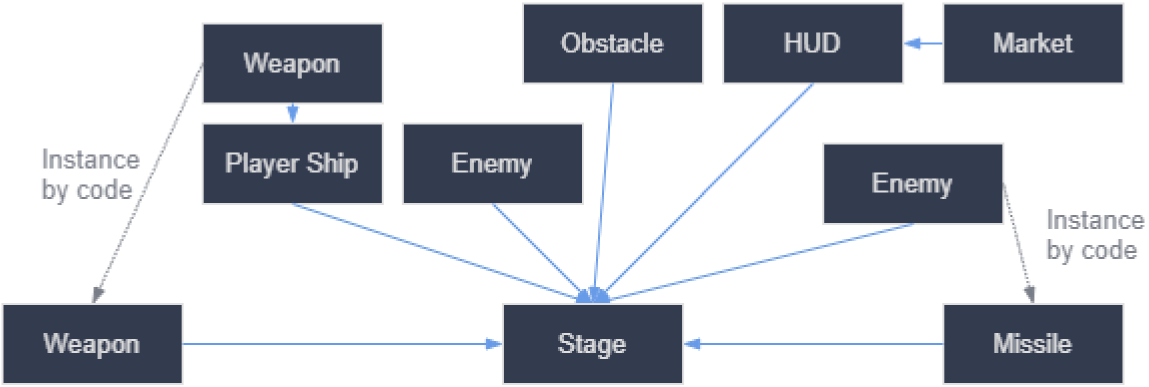
Resources are another essential building block of Godot games we will cover in a later lesson.

Scene instances as a design language

Instances and scenes in Godot offer an excellent design language, setting the engine apart from others out there. We designed Godot around this concept from the ground up.

We recommend dismissing architectural code patterns when making games with Godot, such as Model-View-Controller (MVC) or Entity-Relationship diagrams. Instead, you can start by imagining the elements players will see in your game and structure your code around them.

For example, you could break down a shooter game like so:

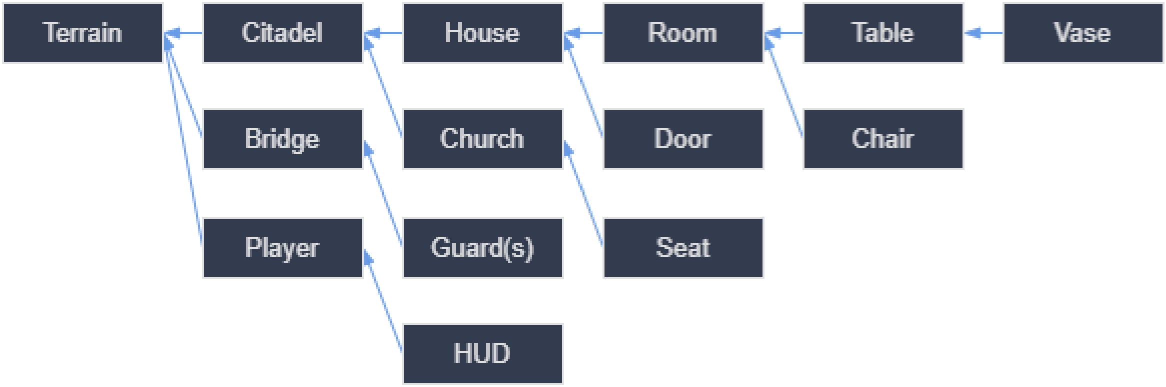

Weapon
Obstacle
HUD
Market
Instance by code
Player Ship
Enemy
Enemy
Instance by code
Weapon
Stage
Missile

You can come up with a diagram like this for almost any type of game. Each rectangle represents an entity that's visible in the game from the player's perspective. The arrows tell you which scene owns which.

Once you have a diagram, we recommend creating a scene for each element listed in it to develop your game. You'll use instancing, either by code or directly in the editor, to build your tree of scenes.

Programmers tend to spend a lot of time designing abstract architectures and trying to fit components into it. Designing based on scenes makes development faster and more straightforward, allowing you to focus on the game logic itself. Because most game components map directly to a scene, using a design based on scene instantiation means you need little other architectural code.

Here's the example of a scene diagram for an open-world game with tons of assets and nested elements:



Imagine we started by creating the room. We could make a couple of different room scenes, with unique arrangements of furniture in them. Later, we could make a house scene that uses multiple room instances for the interior. We would create a citadel out of many instanced houses and a large terrain on which we would place the citadel. Each of these would be a scene instancing one or more sub-scenes.

Later, we could create scenes representing guards and add them to the citadel. They would be indirectly added to the overall game world.

With Godot, it's easy to iterate on your game like this, as all you need to do is create and instantiate more scenes. We designed the editor to be accessible to programmers, designers, and artists alike. A typical team development process can involve 2D or 3D artists, level designers, game designers, and animators, all working with the Godot editor.

Summary

Instancing, the process of producing an object from a blueprint, has many handy uses. With scenes, it gives you:

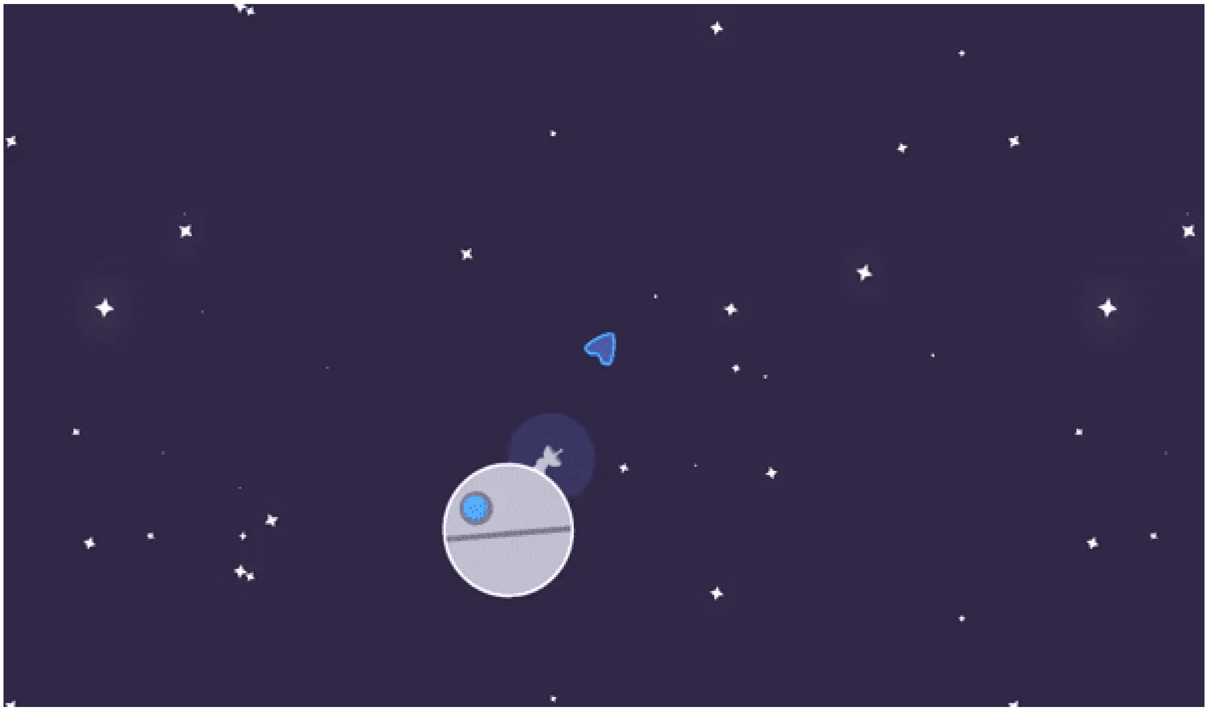
- The ability to divide your game into reusable components.
- A tool to structure and encapsulate complex systems.
- A language to think about your game project's structure in a natural way.

Scripting languages

This lesson will give you an overview of the available scripting languages in Godot. You will learn the pros and cons of each option. In the next part, you will write your first script using GDScript.

Scripts attach to a node and extend its behavior. This means that scripts inherit all functions and properties of the node they attach to.

For example, take a game where a Camera2D node follows a ship. The Camera2D node follows its parent by default. Imagine you want the camera to shake when the player takes damage. As this feature is not built into Godot, you would attach a script to the Camera2D node and code the shake.



Available scripting languages

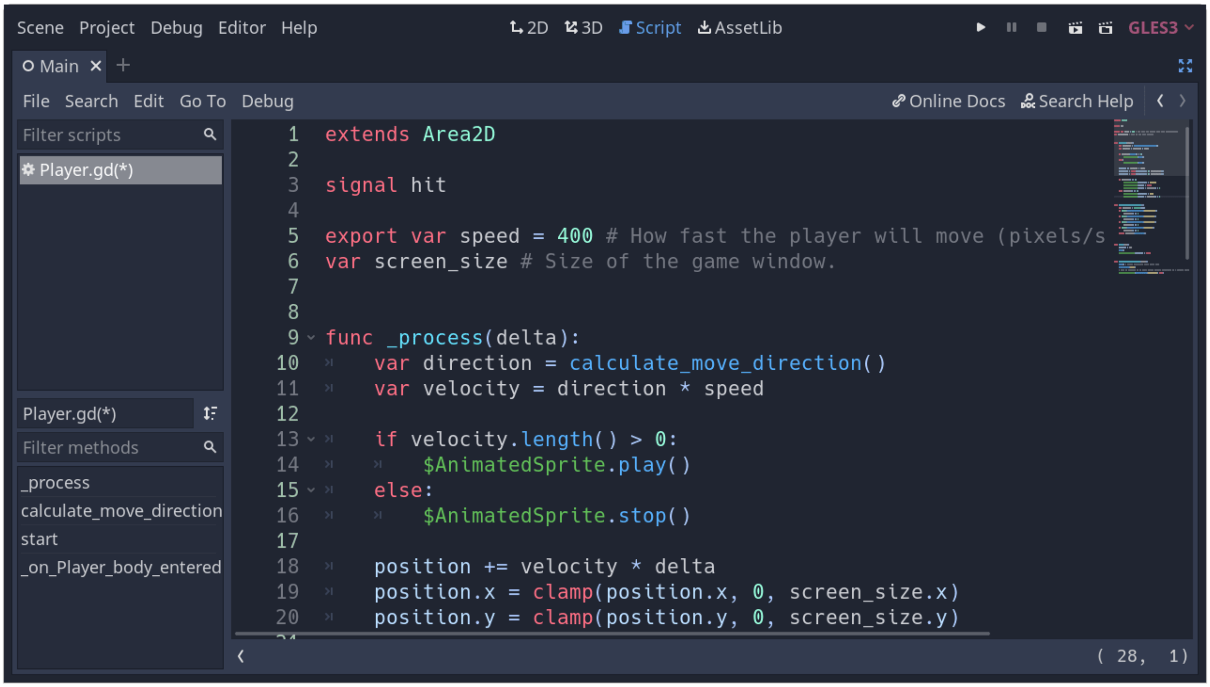
Godot offers **four gameplay programming languages**: GDScript, C#, and, via its GDExtension technology, C and C++. There are more community-supported languages, but these are the official ones.

You can use multiple languages in a single project. For instance, in a team, you could code gameplay logic in GDScript as it's fast to write, and use C# or C++ to implement complex algorithms and maximize their performance. Or you can write everything in GDScript or C#. It's your call.

We provide this flexibility to answer the needs of different game projects and developers.

Which language should I use?

If you're a beginner, we recommend to **start with GDScript**. We made this language specifically for Godot and the needs of game developers. It has a lightweight and straightforward syntax and provides the tightest integration with Godot.


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
GLES3
Main
File
Search
Edit
Go To
Debug
Online Docs
Search Help
Filter scripts
Player.gd(*)
Player.gd(*)
Filter methods
_process
calculate_move_direction
start
_on_Player_body_entered
(28, 1)

```
extends Area2D

signal hit

export var speed = 400 # How fast the player will move (pixels/s
var screen_size # Size of the game window.


func _process(delta):
    var direction = calculate_move_direction()
    var velocity = direction * speed

    if velocity.length() > 0:
        $AnimatedSprite.play()
    else:
        $AnimatedSprite.stop()

    position += velocity * delta
    position.x = clamp(position.x, 0, screen_size.x)
    position.y = clamp(position.y, 0, screen_size.y)
```

For C#, you will need an external code editor like VSCode [https://code.visualstudio.com/] or Visual Studio. While C# support is now mature, you will find fewer learning resources for it compared to GDScript. That's why we recommend C# mainly to users who already have experience with the language.

Let's look at each language's features, as well as its pros and cons.

GDScript

GDScript is an object-oriented [https://en.wikipedia.org/wiki/Object-oriented_programming] and imperative [https://en.wikipedia.org/wiki/Imperative_programming] programming language built for Godot. It's made by and for game developers to save you time coding games. Its features include:

- A simple syntax that leads to short files.
- Blazing fast compilation and loading times.
- Tight editor integration, with code completion for nodes, signals, and more information from the scene it's attached to.

- Built-in vector and transform types, making it efficient for heavy use of linear algebra, a must for games.
- Supports multiple threads as efficiently as statically typed languages.
- No garbage collection [https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)], as this feature eventually gets in the way when creating games. The engine counts references and manages the memory for you in most cases by default, but you can also control memory if you need to.
- Gradual typing [https://en.wikipedia.org/wiki/Gradual_typing]. Variables have dynamic types by default, but you also can use type hints for strong type checks.

GDScript looks like Python as you structure your code blocks using indentations, but it doesn't work the same way in practice. It's inspired by multiple languages, including Squirrel, Lua, and Python.

Note

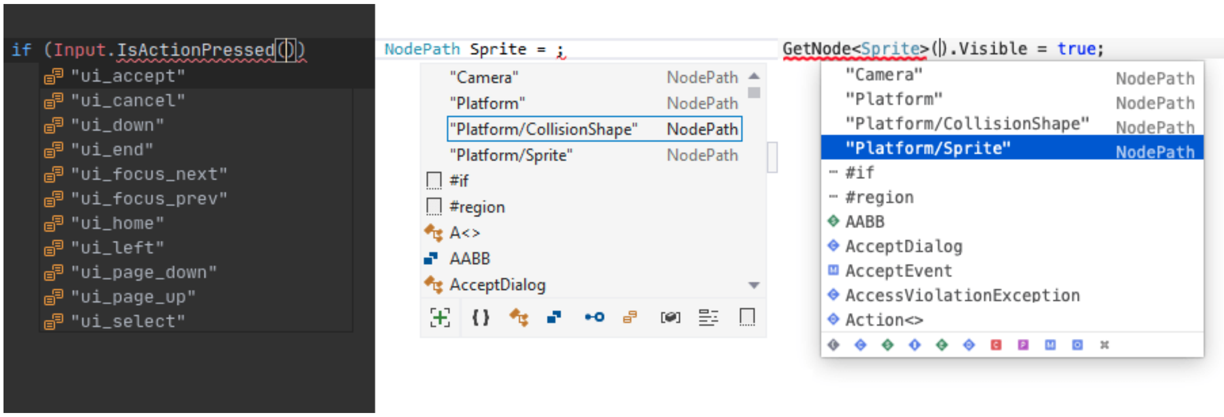
Why don't we use Python or Lua directly?

Years ago, Godot used Python, then Lua. Both languages' integration took a lot of work and had severe limitations. For example, threading support was a big challenge with Python.

Developing a dedicated language doesn't take us more work and we can tailor it to game developers' needs. We're now working on performance optimizations and features that would've been difficult to offer with third-party languages.

.NET / C#

As Microsoft's C# [https://en.wikipedia.org/wiki/C_Sharp_(programming_language)] is a favorite amongst game developers, we officially support it. C# is a mature and flexible language with tons of libraries written for it. We were able to add support for it thanks to a generous donation from Microsoft.



```
if (Input.IsActionPressed())
```

```
NodePath Sprite = ;
```

```
GetNode<Sprite>().Visible = true;
```

"ui_accept"
"ui_cancel"
"ui_down"
"ui_end"
"ui_focus_next"
"ui_focus_prev"
"ui_home"
"ui_left"
"ui_page_down"
"ui_page_up"
"ui_select"
"Camera" NodePath
"Platform" NodePath
"Platform/CollisionShape" NodePath
"Platform/Sprite" NodePath
#if
#region
A<>
AABB
AcceptDialog
"Camera" NodePath
"Platform" NodePath
"Platform/CollisionShape" NodePath
"Platform/Sprite" NodePath
#if
#region
AABB
AcceptDialog
AcceptEvent
AccessViolationException
Action<>

C# offers a good tradeoff between performance and ease of use, although you should be aware of its garbage collector.

Note

You must use the .NET edition of the Godot editor to script in C#. You can download it on the Godot website's download [https://godotengine.org/download/] page.

Since Godot uses .NET 6, in theory, you can use any third-party .NET library or framework in Godot, as well as any Common Language Infrastructure-compliant programming language, such as F#, Boo, or ClojureCLR. However, C# is the only officially supported .NET option.

Note

GDScript code itself doesn't execute as fast as compiled C# or C++. However, most script code calls functions written with fast algorithms in C++ code inside the engine. In many cases, writing gameplay logic in GDScript, C#, or C++ won't have a significant impact on performance.

Attention

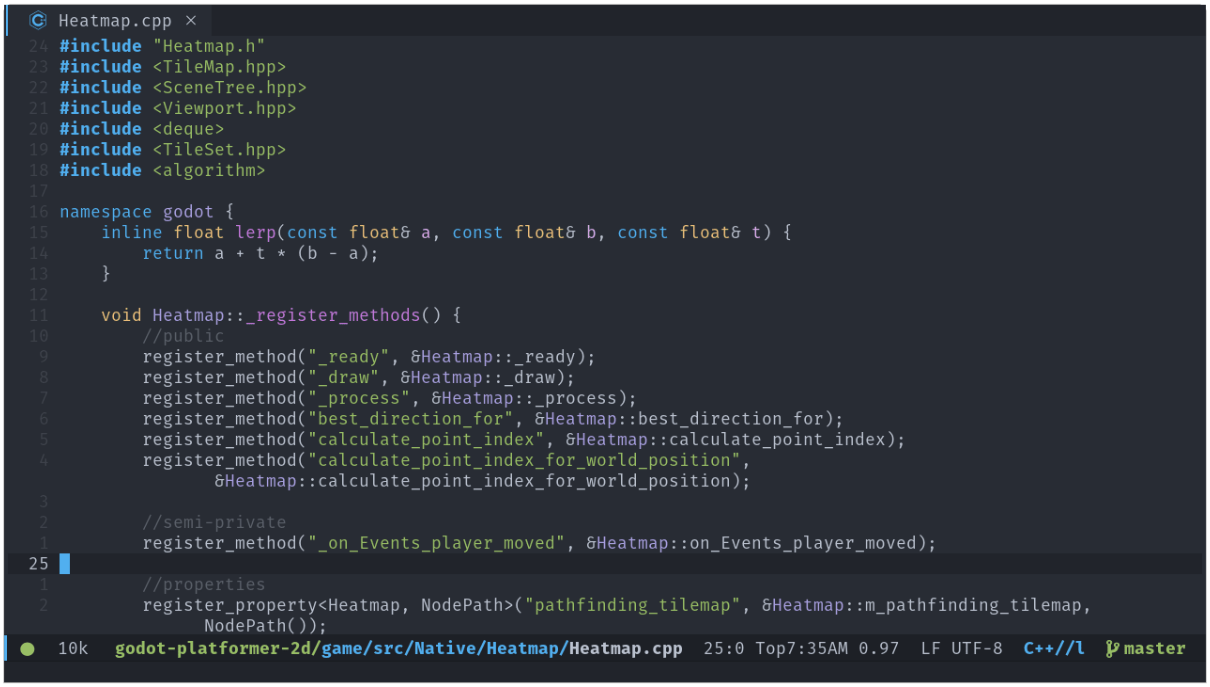
Projects written in C# using Godot 4 currently cannot be exported to the web platform. To use C# on that platform, consider Godot 3 instead. Android and iOS platform support is available as of Godot 4.2, but is experimental and some limitations apply.

See also

To learn more about C#, head to the C# basics page.

C++ via GDExtension

GDExtension allows you to write game code in C++ without needing to recompile Godot.



```cpp
#include "Heatmap.h"
#include <TileMap.hpp>
#include <SceneTree.hpp>
#include <Viewport.hpp>
#include <deque>
#include <TileSet.hpp>
#include <algorithm>

namespace godot {
    inline float lerp(const float& a, const float& b, const float& t) {
        return a + t * (b - a);
    }

    void Heatmap::_register_methods() {
        //public
        register_method("_ready", &Heatmap::_ready);
        register_method("_draw", &Heatmap::_draw);
        register_method("_process", &Heatmap::_process);
        register_method("best_direction_for", &Heatmap::best_direction_for);
        register_method("calculate_point_index", &Heatmap::calculate_point_index);
        register_method("calculate_point_index_for_world_position",
               &Heatmap::calculate_point_index_for_world_position);

        //semi-private
        register_method("_on_Events_player_moved", &Heatmap::on_Events_player_moved);


        //properties
        register_property<Heatmap, NodePath>("pathfinding_tilemap", &Heatmap::m_pathfinding_tilemap,
              NodePath());
```

You can use any version of the language or mix compiler brands and versions for the generated shared libraries, thanks to our use of an internal C API Bridge.

GDExtension is the best choice for performance. You don't need to use it throughout an entire game, as you can write other parts in GDScript or C#.

When working with GDExtension, the available types, functions, and properties closely resemble Godot's actual C++ API.

Summary

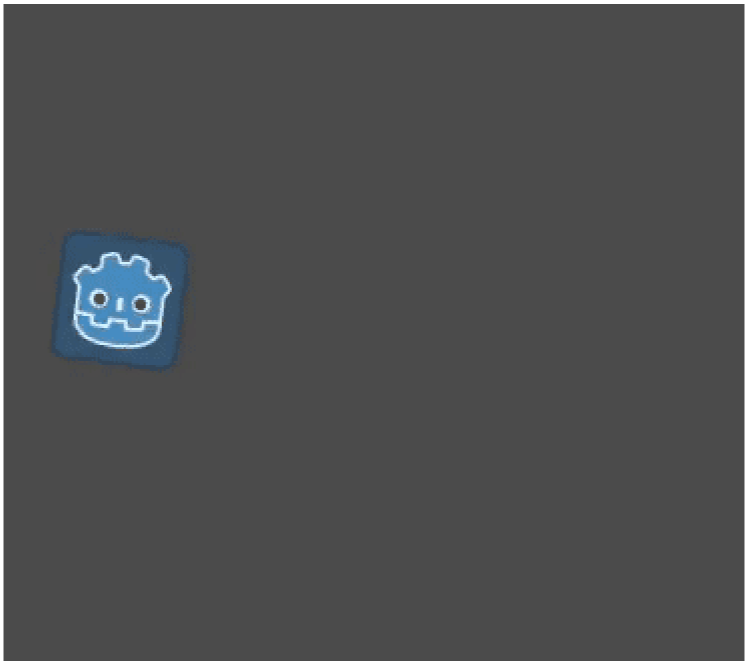
Scripts are files containing code that you attach to a node to extend its functionality.

Godot supports four official scripting languages, offering you flexibility between performance and ease of use.

You can mix languages, for instance, to implement demanding algorithms with C or C++ and write most of the game logic with GDScript or C#.

Creating your first script

In this lesson, you will code your first script to make the Godot icon turn in circles using GDScript. As we mentioned in the introduction, we assume you have programming foundations. The equivalent C# code has been included in another tab for convenience.



See also

To learn more about GDScript, its keywords, and its syntax, head to the GDScript reference.

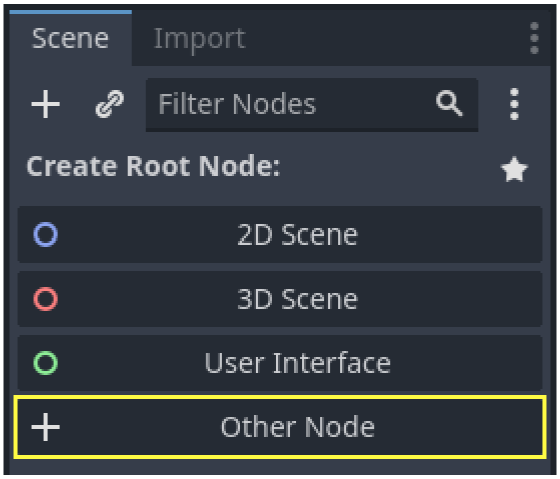
See also

To learn more about C#, head to the C# basics page.

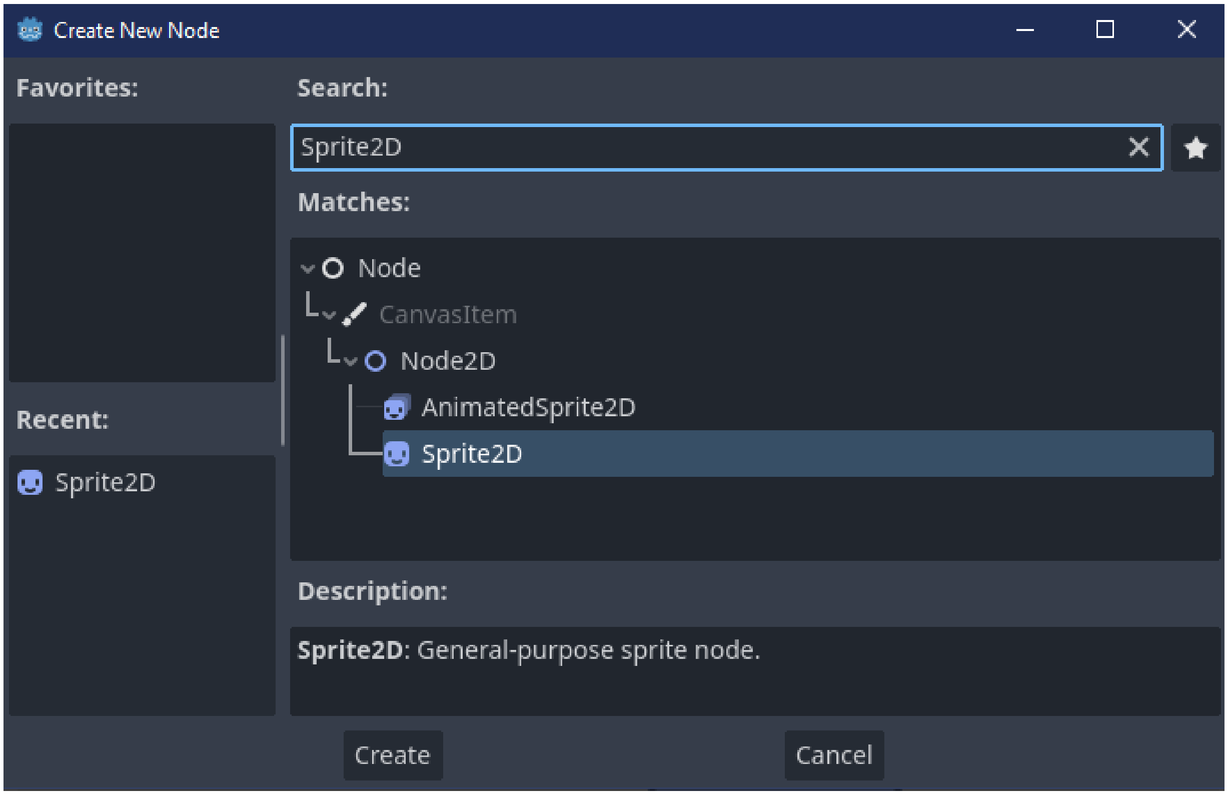
Project setup

Please create a new project to start with a clean slate. Your project should contain one picture: the Godot icon, which we often use for prototyping in the community.

We need to create a Sprite2D node to display it in the game. In the Scene dock, click the Other Node button.

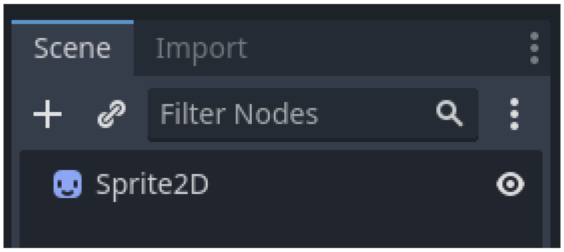

Scene
Import
Filter Nodes
Create Root Node:
2D Scene
3D Scene
User Interface
Other Node

Type "Sprite2D" in the search bar to filter nodes and double-click on Sprite2D to create the node.

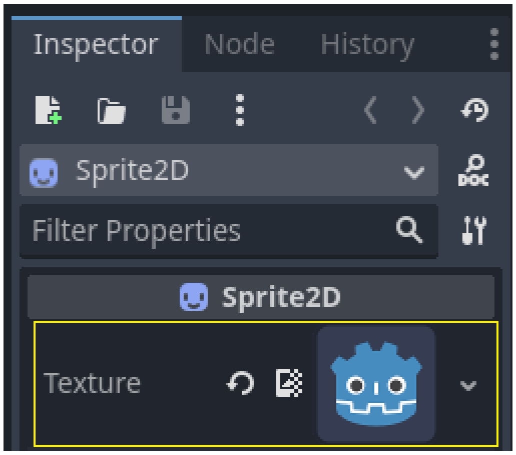

Create New Node
Favorites:
Search:
Sprite2D
Matches:
Node
CanvasItem
Node2D
AnimatedSprite2D
Sprite2D
Recent:
Sprite2D
Description:
Create
Cancel

Sprite2D: General-purpose sprite node.

Your Scene tab should now only have a Sprite2D node.


Scene
Import
Filter Nodes
Sprite2D

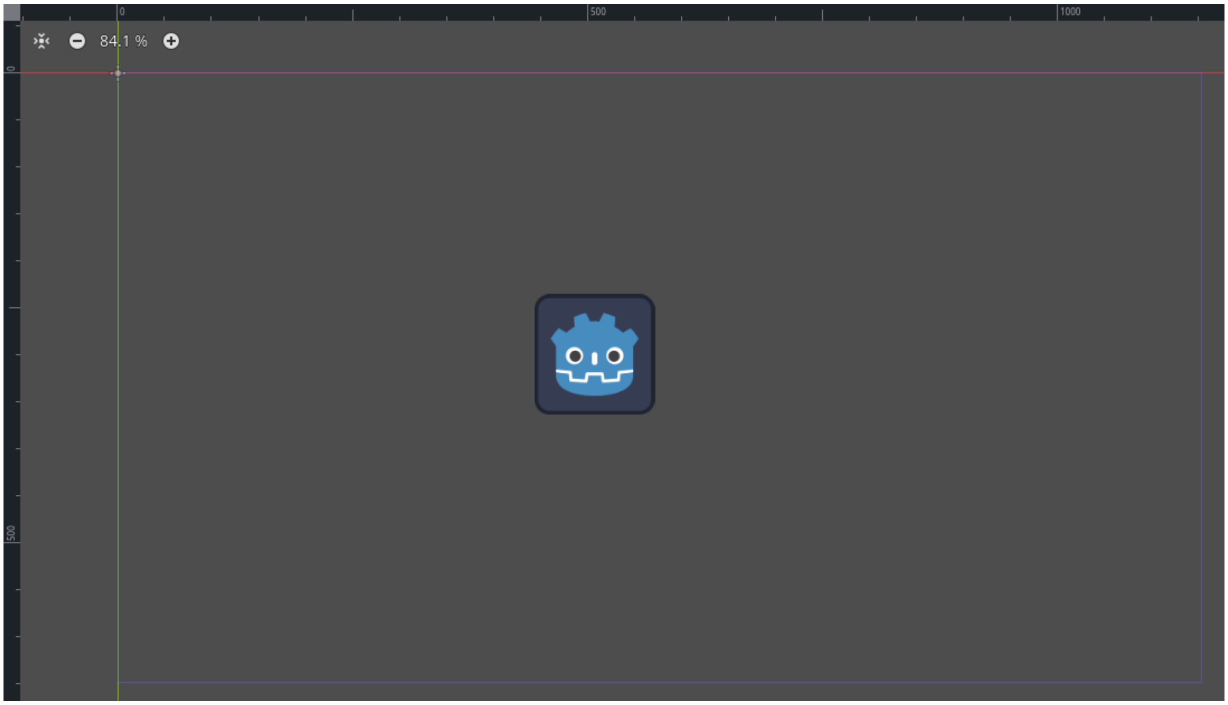
A Sprite2D node needs a texture to display. In the Inspector on the right, you can see that the Texture property says "[empty]". To display the Godot icon, click and drag the file `icon.svg` from the FileSystem dock onto the Texture slot.


Inspector
Node
History
Sprite2D
Filter Properties
Sprite2D
Texture

Note

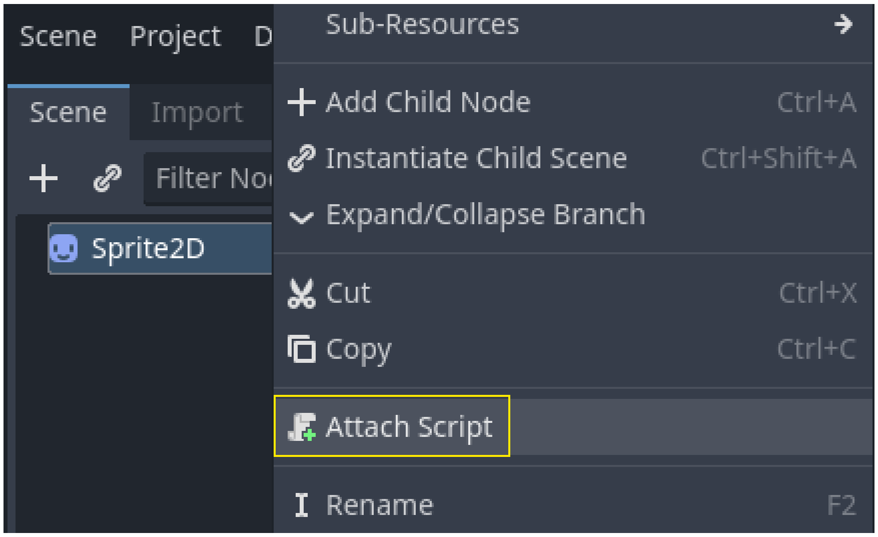
You can create Sprite2D nodes automatically by dragging and dropping images on the viewport.

Then, click and drag the icon in the viewport to center it in the game view.


0
500
1000
84.1 %
0
500

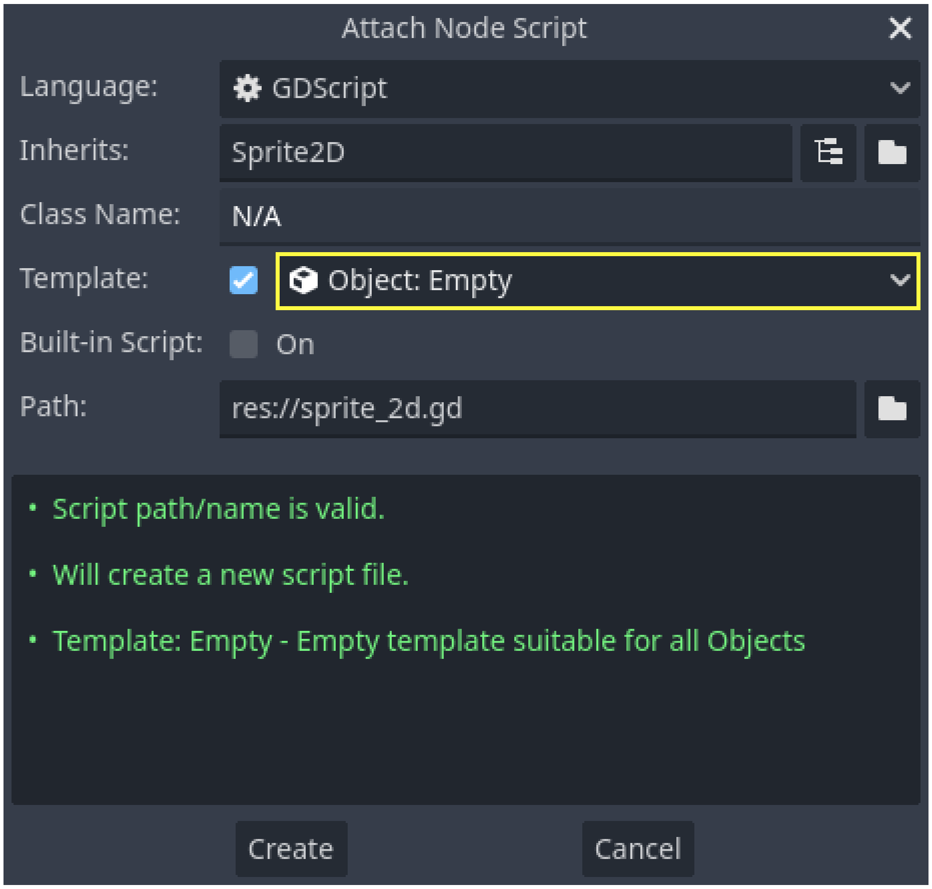
Creating a new script

To create and attach a new script to our node, right-click on Sprite2D in the scene dock and select "Attach Script".


Scene
Project
D
Scene
Import
Filter No
Sprite2D
Sub-Resources
Add Child Node Ctrl+A
Instantiate Child Scene Ctrl+Shift+A
Expand/Collapse Branch
Cut Ctrl+X
Copy Ctrl+C
Attach Script
Rename F2

The Attach Node Script window appears. It allows you to select the script's language and file path, among other options.

Change the Template field from "Node: Default" to "Object: Empty" to start with a clean file. Leave the other options set to their default values and click the Create button to create the script.



Language: GDScript

Inherits: Sprite2D

Class Name: N/A

Template: ☑ Object: Empty

Built-in Script: ☐ On

Path: res://sprite_2d.gd

- Script path/name is valid.
- Will create a new script file.
- Template: Empty - Empty template suitable for all Objects

Note

C# script names need to match their class name. In this case, you should name the file `MySprite2D.cs`.

The Script workspace should appear with your new `sprite_2d.gd` file open and the following line of code:

GDScript

```
extends Sprite2D
```

C#

```
using Godot;
using System;
```

```
public partial class MySprite2D : Sprite2D
{
}
```

Every GDScript file is implicitly a class. The `extends` keyword defines the class this script inherits or extends. In this case, it's `Sprite2D`, meaning our script will get access to all the properties and functions of the Sprite2D node, including classes it extends, like `Node2D`, `CanvasItem`, and `Node`.

Note

In GDScript, if you omit the line with the `extends` keyword, your class will implicitly extend RefCounted, which Godot uses to manage your application's memory.

Inherited properties include the ones you can see in the Inspector dock, like our node's `texture`.

Note

By default, the Inspector displays a node's properties in "Title Case", with capitalized words separated by a space. In GDScript code, these properties are in "snake_case", which is lowercase with words separated by an underscore.

You can hover over any property's name in the Inspector to see a description and its identifier in code.

Hello, world!

Our script currently doesn't do anything. Let's make it print the text "Hello, world!" to the Output bottom panel to get started.

Add the following code to your script:

GDScript

```
func _init():
    print("Hello, world!")
```

C#

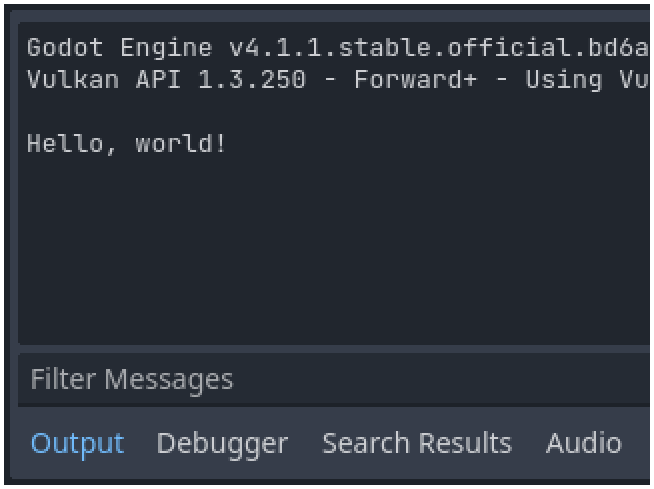
```
public MySprite2D()
{
    GD.Print("Hello, world!");
}
```

Let's break it down. The `func` keyword defines a new function named `_init`. This is a special name for our class's constructor. The engine calls `_init()` on every object or node upon creating it in memory, if you define this function.

Note

GDScript is an indent-based language. The tab at the start of the line that says `print()` is necessary for the code to work. If you omit it or don't indent a line correctly, the editor will highlight it in red and display the following error message: "Indented block expected".

Save the scene as `sprite_2d.tscn` if you haven't already, then press `F6` (`Cmd + R` on macOS) to run it. Look at the **Output** bottom panel that expands. It should display "Hello, world!".



```
Godot Engine v4.1.1.stable.official.bd6a
Vulkan API 1.3.250 - Forward+ - Using Vu

Hello, world!
```

Delete the `_init()` function, so you're only left with the line `extends Sprite2D`.

Turning around

It's time to make our node move and rotate. To do so, we're going to add two member variables to our script: the movement speed in pixels per second and the angular speed in radians per second. Add the following after the `extends Sprite2D` line.

GDScript

```
var speed = 400
var angular_speed = PI
```

C#

```
private int _speed = 400;
private float _angularSpeed = Mathf.Pi;
```

Member variables sit near the top of the script, after any "extends" lines, but before functions. Every node instance with this script attached to it will have its own copy of the `speed` and `angular_speed` properties.

Note

Angles in Godot work in radians by default, but you have built-in functions and properties available if you prefer to calculate angles in degrees instead.

To move our icon, we need to update its position and rotation every frame in the game loop. We can use the `_process()` virtual function of the `Node` class. If you define it in any class that extends the Node class, like Sprite2D, Godot will call the function every frame and pass it an argument named `delta`, the time elapsed since the last frame.

Note

Games work by rendering many images per second, each called a frame, and they do so in a loop. We measure the rate at which a game produces images in Frames Per Second (FPS). Most games aim for 60 FPS, although you might find figures like 30 FPS on slower mobile devices or 90 to 240 for virtual reality games.

The engine and game developers do their best to update the game world and render images at a constant time interval, but there are always small variations in frame render times. That's why the engine provides us with this delta time value, making our motion independent of our framerate.

At the bottom of the script, define the function:

GDScript

```
func _process(delta):
    rotation += angular_speed * delta
```

C#

```
public override void _Process(double delta)
{
    Rotation += _angularSpeed * (float)delta;
}
```

The `func` keyword defines a new function. After it, we have to write the function's name and arguments it takes in parentheses. A colon ends the definition, and the indented blocks that follow are the function's content or instructions.

Note

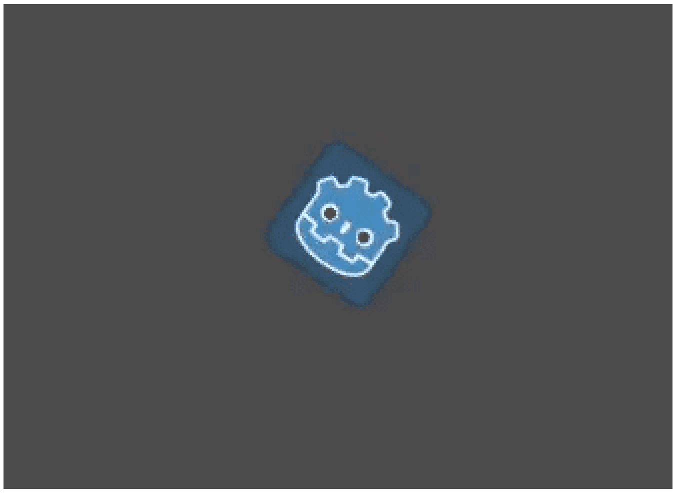
Notice how `_process()`, like `_init()`, starts with a leading underscore. By convention, Godot's virtual functions, that is to say, built-in functions you can override to communicate with the engine, start with an underscore.

The line inside the function, `rotation += angular_speed * delta`, increments our sprite's rotation every frame. Here, `rotation` is a property inherited from the class `Node2D`, which `Sprite2D` extends. It controls the rotation of our node and works with radians.

Tip

In the code editor, you can Ctrl-click (Cmd-click on MacOS) on any built-in property or function like `position`, `rotation`, or `_process` to open the corresponding documentation in a new tab.

Run the scene to see the Godot icon turn in-place.



Note

In C#, notice how the `delta` argument taken by `_Process()` is a `double`. We therefore need to convert it to `float` when we apply it to the rotation.

Moving forward

Let's now make the node move. Add the following two lines inside of the `_process()` function, ensuring the new lines are indented the same way as the `rotation += angular_speed * delta` line before them.

GDScript

```
var velocity = Vector2.UP.rotated(rotation) * speed

position += velocity * delta
```

C#

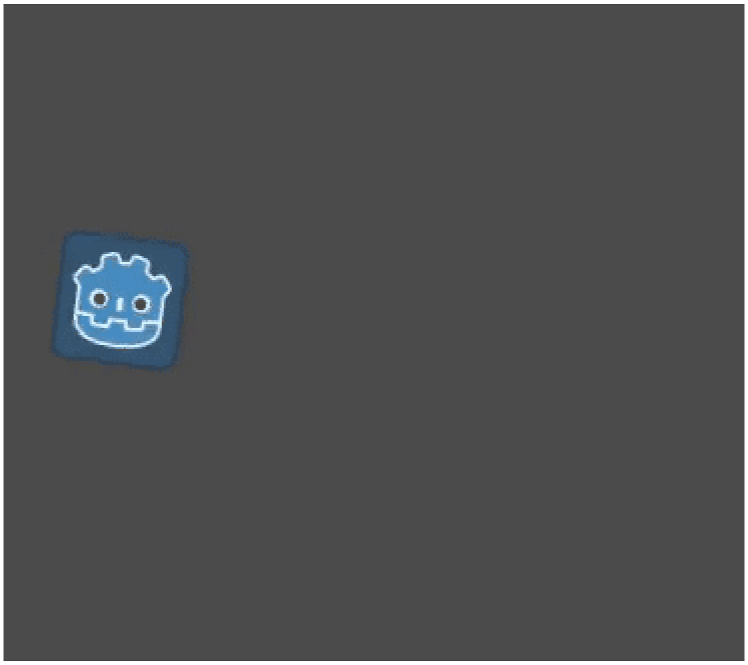
```
var velocity = Vector2.Up.Rotated(Rotation) * _speed;

Position += velocity * (float)delta;
```

As we already saw, the `var` keyword defines a new variable. If you put it at the top of the script, it defines a property of the class. Inside a function, it defines a local variable: it only exists within the function's scope.

We define a local variable named `velocity`, a 2D vector representing both a direction and a speed. To make the node move forward, we start from the Vector2 class's constant `Vector2.UP`, a vector pointing up, and rotate it by calling the Vector2 method `rotated()`. This expression, `Vector2.UP.rotated(rotation)`, is a vector pointing forward relative to our icon. Multiplied by our `speed` property, it gives us a velocity we can use to move the node forward.

We add `velocity * delta` to the node's `position` to move it. The position itself is of type Vector2, a built-in type in Godot representing a 2D vector.

Run the scene to see the Godot head run in circles.



Note

Moving a node like that does not take into account colliding with walls or the floor. In Your first 2D game, you will learn another approach to moving objects while detecting collisions.

Our node currently moves by itself. In the next part, Listening to player input, we'll use player input to control it.

Complete script

Here is the complete `sprite_2d.gd` file for reference.

GDScript

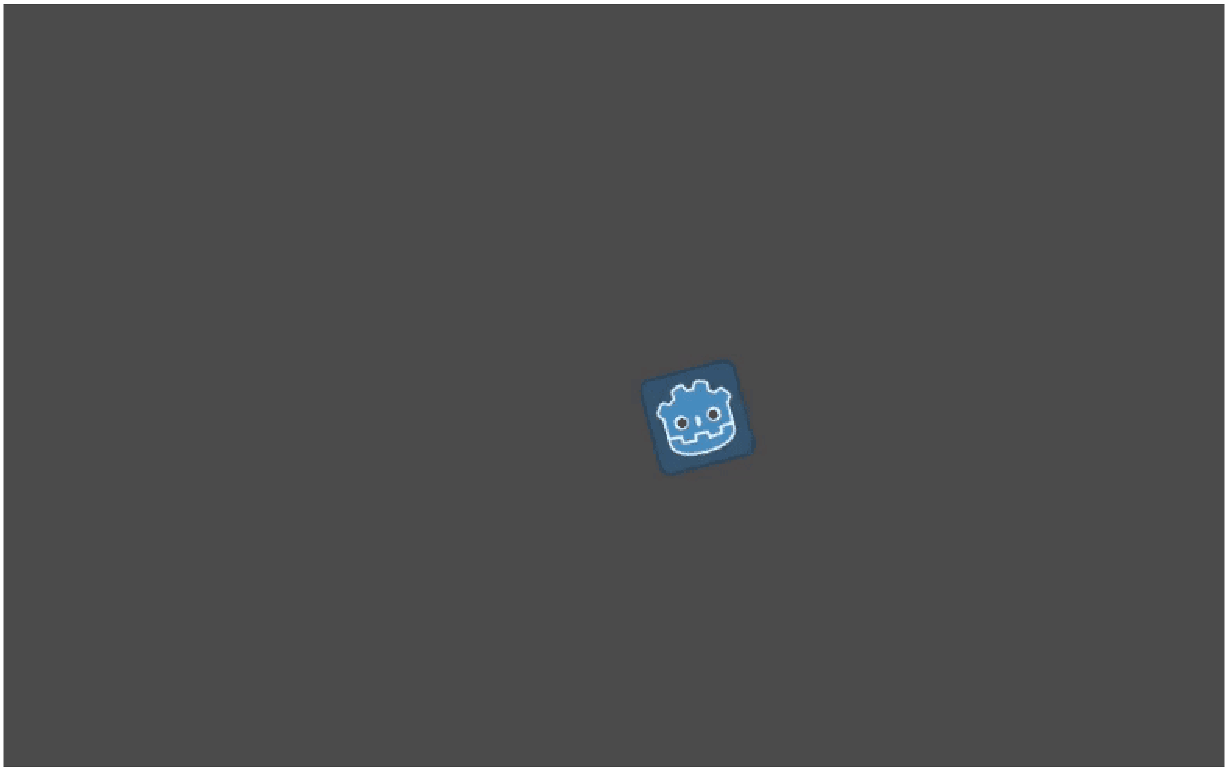
```
extends Sprite2D

var speed = 400
var angular_speed = PI


func _process(delta):
    rotation += angular_speed * delta

    var velocity = Vector2.UP.rotated(rotation) * speed

    position += velocity * delta
```

C#

```
using Godot;
using System;

public partial class MySprite2D : Sprite2D
{
    private int _speed = 400;
    private float _angularSpeed = Mathf.Pi;

    public override void _Process(double delta)
    {
        Rotation += _angularSpeed * (float)delta;
        var velocity = Vector2.Up.Rotated(Rotation) * _speed;

        Position += velocity * (float)delta;
    }
}
```

Listening to player input

Building upon the previous lesson, Creating your first script, let's look at another important feature of any game: giving control to the player. To add this, we need to modify our `sprite_2d.gd` code.



You have two main tools to process the player's input in Godot:

1. The built-in input callbacks, mainly `_unhandled_input()`. Like `_process()`, it's a built-in virtual function that Godot calls every time the player presses a key. It's the tool you want to use to react to events that don't happen every frame, like pressing `Space` to jump. To learn more about input callbacks, see Using InputEvent.
2. The `Input` singleton. A singleton is a globally accessible object. Godot provides access to several in scripts. It's the right tool to check for input every frame.

We're going to use the `Input` singleton here as we need to know if the player wants to turn or move every frame.

For turning, we should use a new variable: `direction`. In our `_process()` function, replace the `rotation += angular_speed * delta` line with the code below.

GDScript

```
var direction = 0
if Input.is_action_pressed("ui_left"):
    direction = -1
if Input.is_action_pressed("ui_right"):
    direction = 1

rotation += angular_speed * direction * delta
```

C#

```
var direction = 0;
if (Input.IsActionPressed("ui_left"))
{
    direction = -1;
}
if (Input.IsActionPressed("ui_right"))
{
    direction = 1;
}

Rotation += _angularSpeed * direction * (float)delta;
```

Our `direction` local variable is a multiplier representing the direction in which the player wants to turn. A value of `0` means the player isn't pressing the left or the right arrow key. A value of `1` means the player wants to turn right, and `-1` means they want to turn left.

To produce these values, we introduce conditions and the use of `Input`. A condition starts with the `if` keyword in GDScript and ends with a colon. The condition is the expression between the keyword and the end of the line.

To check if a key was pressed this frame, we call `Input.is_action_pressed()`. The method takes a text string representing an input action and returns `true` if the action is pressed, `false` otherwise.

The two actions we use above, "ui_left" and "ui_right", are predefined in every Godot project. They respectively trigger when the player presses the left and right arrows on the keyboard or left and right on a gamepad's D-pad.

Note

You can see and edit input actions in your project by going to Project -> Project Settings and clicking on the Input Map tab.

Finally, we use the `direction` as a multiplier when we update the node's `rotation`: `rotation += angular_speed * direction * delta`.

Comment out the lines `var velocity = Vector2.UP.rotated(rotation) * speed` and `position += velocity * delta` like this:

GDScript

```
#var velocity = Vector2.UP.rotated(rotation) * speed

#position += velocity * delta
```

C#

```
//var velocity = Vector2.Up.Rotated(Rotation) * _speed;

//Position += velocity * (float)delta;
```

This will ignore the code that moved the icon's position in a circle without user input from the previous exercise.

If you run the scene with this code, the icon should rotate when you press `Left` and `Right`.

Moving when pressing "up"

To only move when pressing a key, we need to modify the code that calculates the velocity. Uncomment the code and replace the line starting with `var velocity` with the code below.

GDScript

```
var velocity = Vector2.ZERO
if Input.is_action_pressed("ui_up"):
    velocity = Vector2.UP.rotated(rotation) * speed
```

C#

```
var velocity = Vector2.Zero;
if (Input.IsActionPressed("ui_up"))
{
    velocity = Vector2.Up.Rotated(Rotation) * _speed;
}
```

We initialize the `velocity` with a value of `Vector2.ZERO`, another constant of the built-in `Vector` type representing a 2D vector of length 0.

If the player presses the "ui_up" action, we then update the velocity's value, causing the sprite to move forward.

Complete script

Here is the complete `sprite_2d.gd` file for reference.

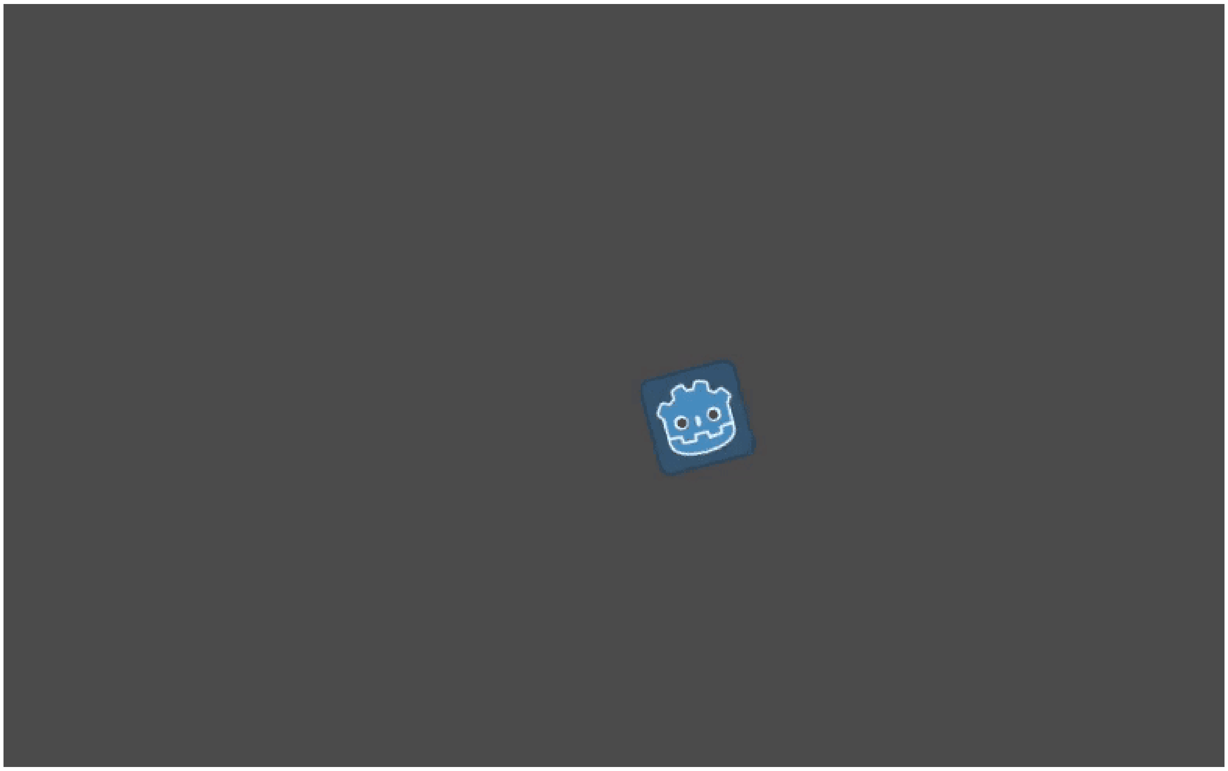
GDScript

```
extends Sprite2D

var speed = 400
var angular_speed = PI


func _process(delta):
    var direction = 0
    if Input.is_action_pressed("ui_left"):
        direction = -1
    if Input.is_action_pressed("ui_right"):
        direction = 1

    rotation += angular_speed * direction * delta

    var velocity = Vector2.ZERO
    if Input.is_action_pressed("ui_up"):
        velocity = Vector2.UP.rotated(rotation) * speed

    position += velocity * delta
```

C#

```
using Godot;

public partial class MySprite2D : Sprite2D
{
    private float _speed = 400;
    private float _angularSpeed = Mathf.Pi;

    public override void _Process(double delta)
    {
        var direction = 0;
        if (Input.IsActionPressed("ui_left"))
        {
            direction = -1;
        }
        if (Input.IsActionPressed("ui_right"))
        {
            direction = 1;
        }

        Rotation += _angularSpeed * direction * (float)delta;

        var velocity = Vector2.Zero;
        if (Input.IsActionPressed("ui_up"))
        {
            velocity = Vector2.Up.Rotated(Rotation) * _speed;
        }

        Position += velocity * (float)delta;
    }
}
```

If you run the scene, you should now be able to rotate with the left and right arrow keys and move forward by pressing `Up`.



Summary

In summary, every script in Godot represents a class and extends one of the engine's built-in classes. The node types your classes inherit from give you access to properties, such as `rotation` and `position` in our sprite's case. You also inherit many functions, which we didn't get to use in this example.

In GDScript, the variables you put at the top of the file are your class's properties, also called member variables. Besides variables, you can define functions, which, for the most part, will be your classes' methods.

Godot provides several virtual functions you can define to connect your class with the engine. These include `_process()`, to apply changes to the node every frame, and `_unhandled_input()`, to receive input events like key and button presses from the users. There are quite a few more.

The `Input` singleton allows you to react to the players' input anywhere in your code. In particular, you'll get to use it in the `_process()` loop.

In the next lesson, Using signals, we'll build upon the relationship between scripts and nodes by having our nodes trigger code in scripts.

Using signals

In this lesson, we will look at signals. They are messages that nodes emit when something specific happens to them, like a button being pressed. Other nodes can connect to that signal and call a function when the event occurs.

Signals are a delegation mechanism built into Godot that allows one game object to react to a change in another without them referencing one another. Using signals limits coupling [https://en.wikipedia.org/wiki/Coupling_(computer_programming)] and keeps your code flexible.

For example, you might have a life bar on the screen that represents the player's health. When the player takes damage or uses a healing potion, you want the bar to reflect the change. To do so, in Godot, you would use signals.

Like methods (Callable), signals are a first-class type since Godot 4.0. This means you can pass them around as method arguments directly without having to pass them as strings, which allows for better autocompletion and is less error-prone. See the Signal class reference for a list of what you can do with the Signal type directly.

See also

As mentioned in the introduction, signals are Godot's version of the observer pattern. You can learn more about it in Game Programming Patterns [https://gameprogrammingpatterns.com/observer.html].

We will now use a signal to make our Godot icon from the previous lesson (Listening to player input) move and stop by pressing a button.

Note

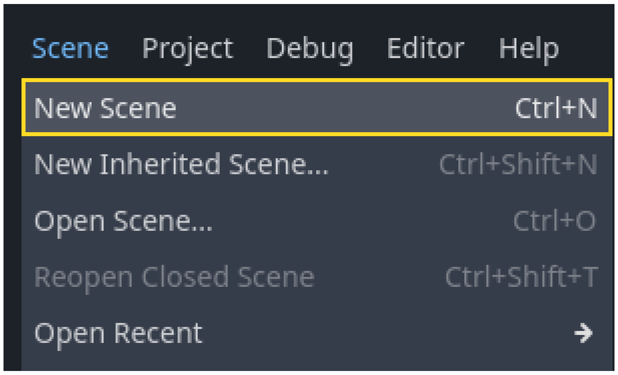
For this project, we will be following the Godot naming conventions.

- **GDScript**: Classes (nodes) use PascalCase, variables and functions use snake_case, and constants use ALL_CAPS (See GDScript style guide).
- **C#**: Classes, export variables and methods use PascalCase, private fields use _camelCase, local variables and parameters use camelCase (See C# style guide). Be careful to type the method names precisely when connecting signals.

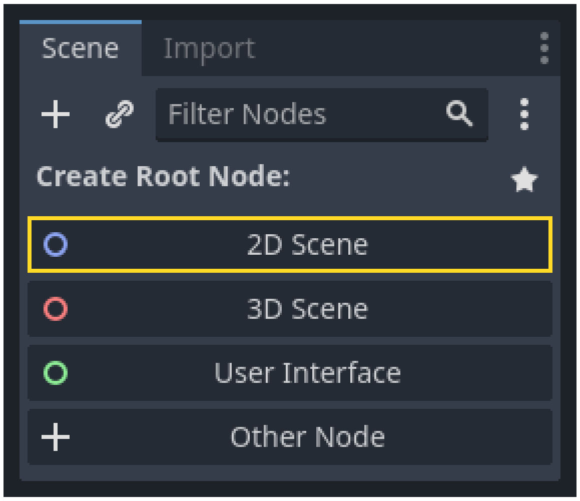
Scene setup

To add a button to our game, we will create a new scene which will include both a Button and the `sprite_2d.tscn` scene we created in the Creating your first script lesson.

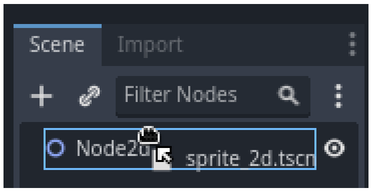
Create a new scene by going to the menu Scene -> New Scene.


Scene
Project
Debug
Editor
Help
New Scene
Ctrl+N
New Inherited Scene...
Ctrl+Shift+N
Open Scene...
Ctrl+O
Reopen Closed Scene
Ctrl+Shift+T
Open Recent

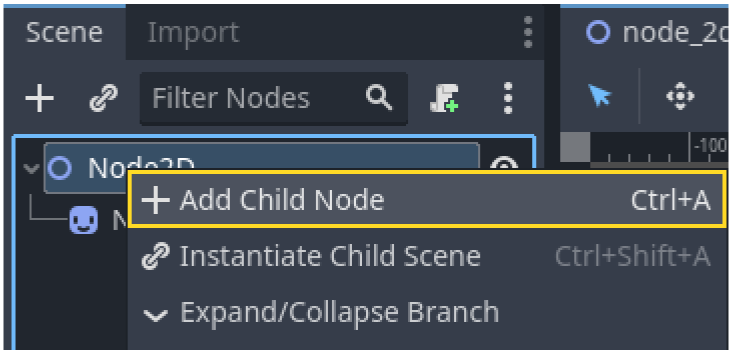
In the Scene dock, click the 2D Scene button. This will add a Node2D as our root.


Scene
Import
Filter Nodes
Create Root Node:
2D Scene
3D Scene
User Interface
Other Node

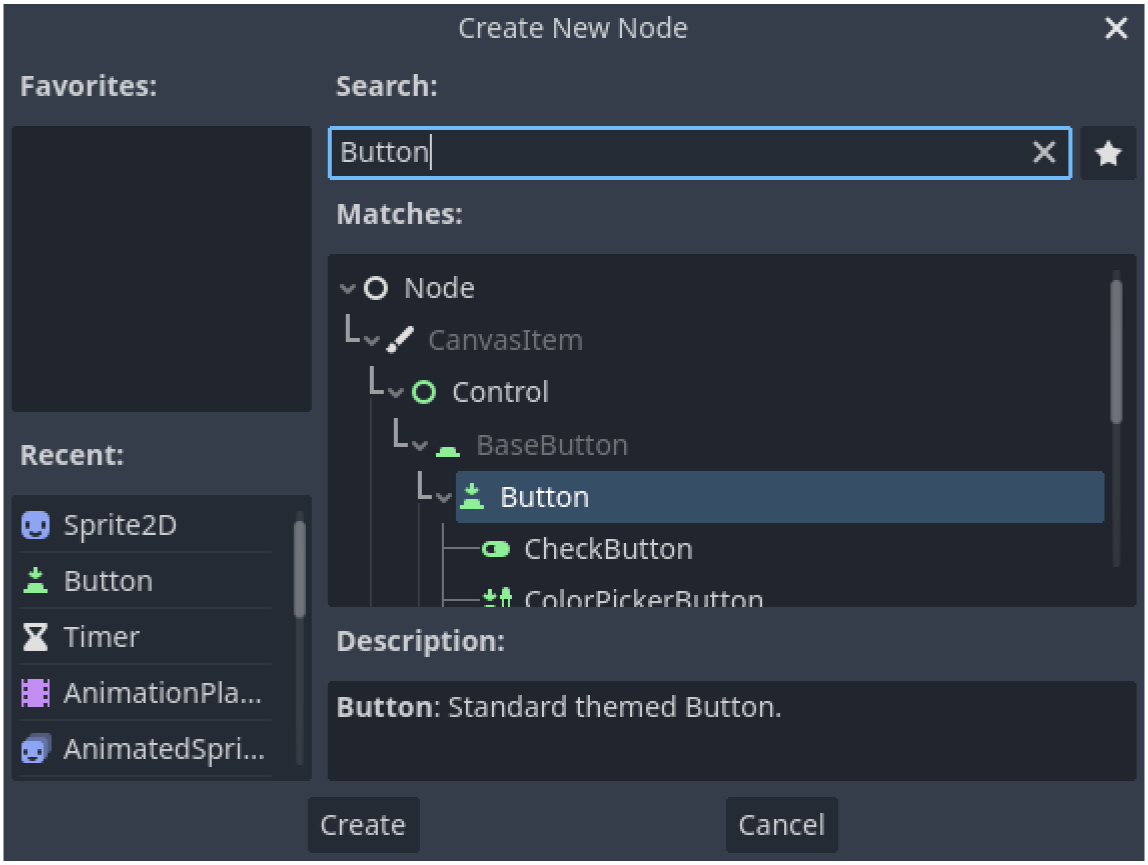
In the FileSystem dock, click and drag the `sprite_2d.tscn` file you saved previously onto the Node2D to instantiate it.


Scene
Import
Filter Nodes
Node2D
sprite_2d.tscn

We want to add another node as a sibling of the Sprite2D. To do so, right-click on Node2D and select Add Child Node.


Scene
Import
Filter Nodes
node_2c
Node2D
Add Child Node
Ctrl+A
Instantiate Child Scene
Ctrl+Shift+A
Expand/Collapse Branch

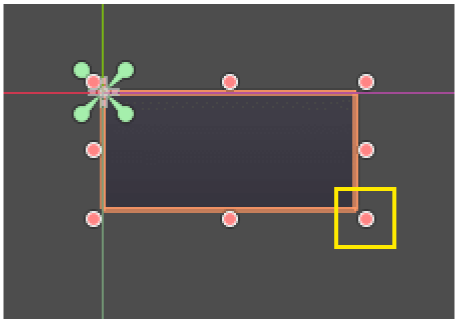
Search for the Button node and add it.


Create New Node
Favorites:
Search:
Matches:
Node
CanvasItem
Control
BaseButton
Button
CheckButton
ColorPickerButton
Recent:
Sprite2D
Button
Timer
AnimationPla...
AnimatedSpri...
Description:
Create
Cancel

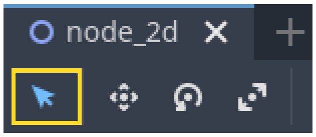
Search: Button

Button: Standard themed Button.

The node is small by default. Click and drag on the bottom-right handle of the Button in the viewport to resize it.

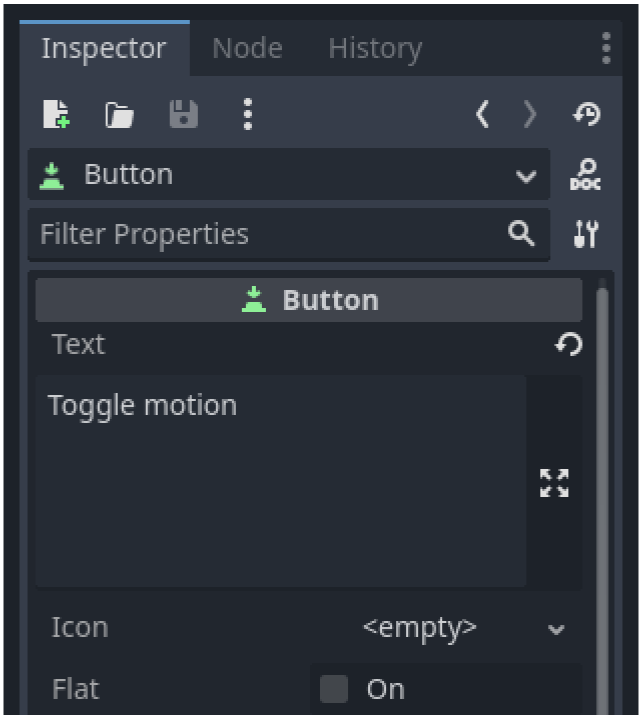


If you don't see the handles, ensure the select tool is active in the toolbar.



Click and drag on the button itself to move it closer to the sprite.

You can also write a label on the Button by editing its Text property in the Inspector. Enter `Toggle motion`.

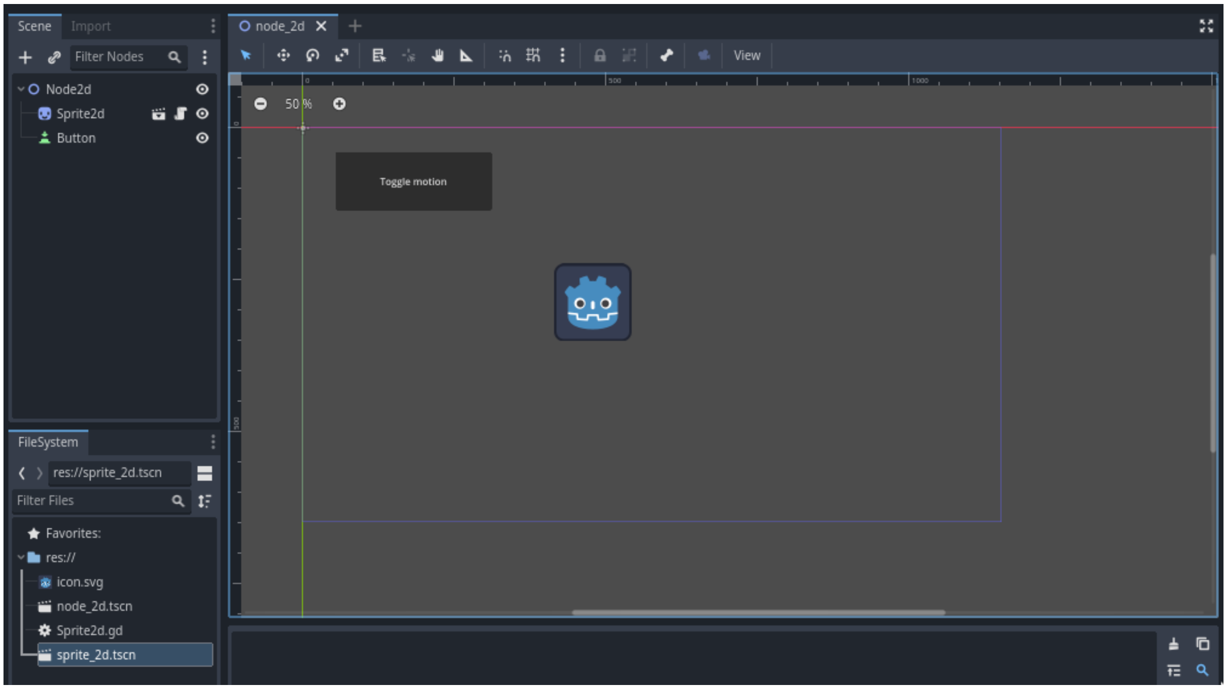

Inspector
Node
History
Button
Filter Properties
Button

Text

Toggle motion

Icon
<empty>
Flat
On

Your scene tree and viewport should look like this.

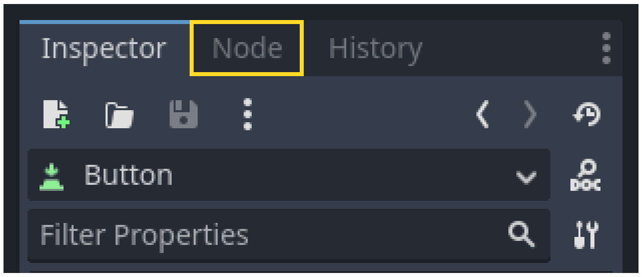

Scene
Import
Filter Nodes
Node2d
Sprite2d
Button
FileSystem
res://sprite_2d.tscn
Filter Files
Favorites:
res://
icon.svg
node_2d.tscn
Sprite2d.gd
sprite_2d.tscn
node_2d
View
50 %
Toggle motion

Save your newly created scene as `node_2d.tscn`, if you haven't already. You can then run it with `F6` (`Cmd + R` on macOS). At the moment, the button will be visible, but nothing will happen if you press it.

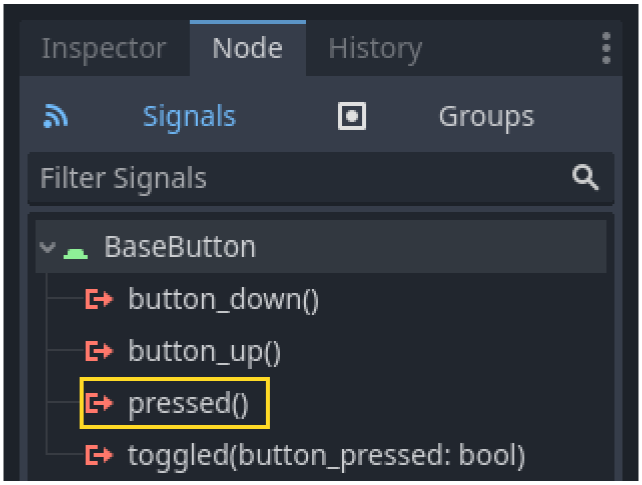
Connecting a signal in the editor

Here, we want to connect the Button's "pressed" signal to our Sprite2D, and we want to call a new function that will toggle its motion on and off. We need to have a script attached to the Sprite2D node, which we do from the previous lesson.

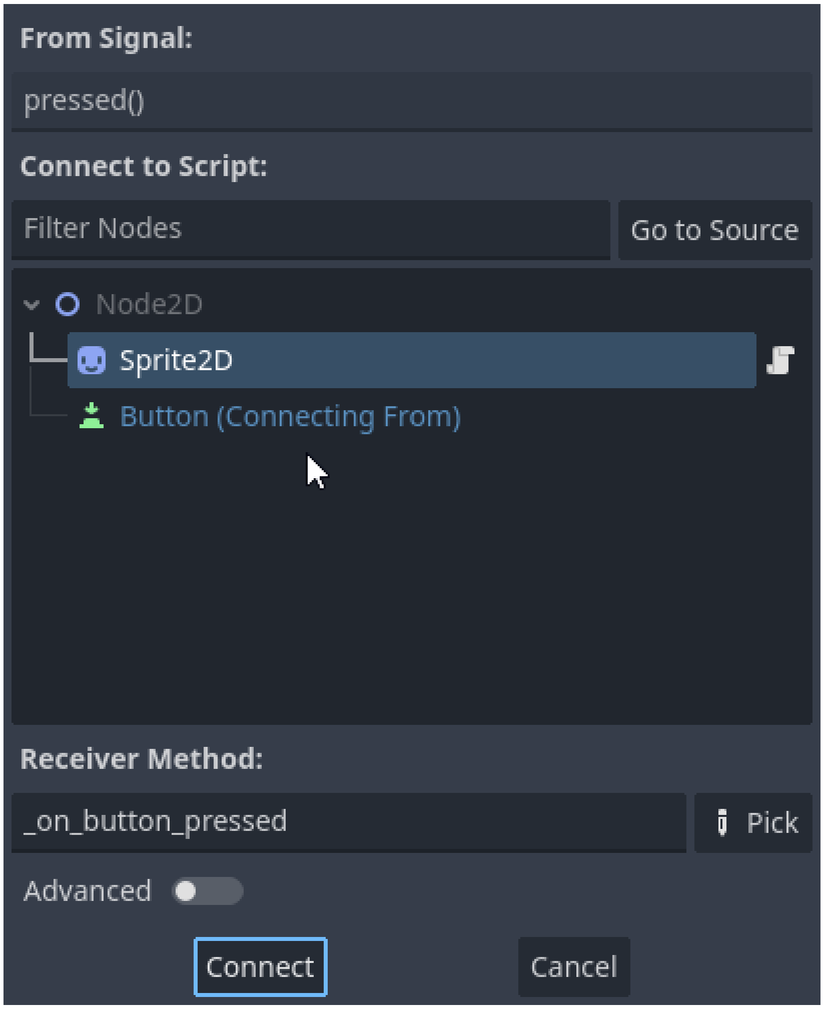
You can connect signals in the Node dock. Select the Button node and, on the right side of the editor, click on the tab named "Node" next to the Inspector.


Inspector
Node
History
Button
Filter Properties

The dock displays a list of signals available on the selected node.


Inspector
Node
History
Signals
Groups
Filter Signals
BaseButton
button_down()
button_up()
pressed()
toggled(button_pressed: bool)

Double-click the "pressed" signal to open the node connection window.



From Signal: pressed()

Connect to Script:

Filter Nodes
Go to Source
Node2D
Sprite2D
Button (Connecting From)

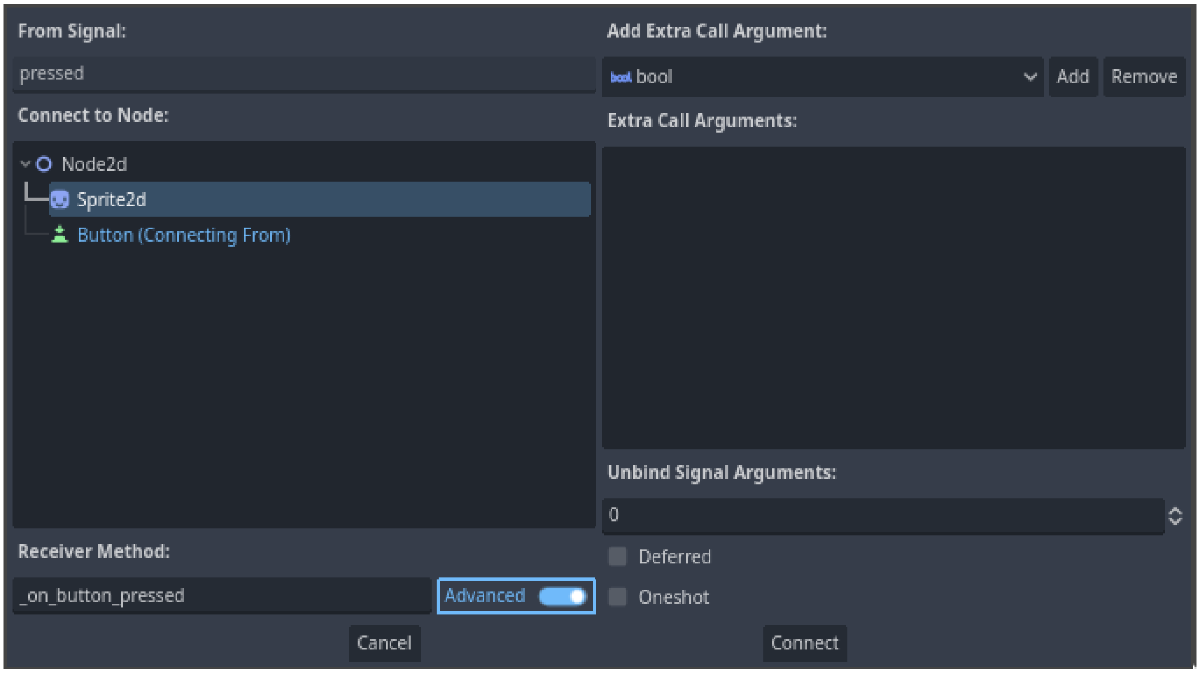
Receiver Method: _on_button_pressed

Pick
Advanced
Connect
Cancel

There, you can connect the signal to the Sprite2D node. The node needs a receiver method, a function that Godot will call when the Button emits the signal. The editor generates one for you. By convention, we name these callback methods "_on_node_name_signal_name". Here, it'll be "_on_button_pressed".

Note

When connecting signals via the editor's Node dock, you can use two modes. The simple one only allows you to connect to nodes that have a script attached to them and creates a new callback function on them.



From Signal: pressed

Connect to Node:
Node2d
Sprite2d
Button (Connecting From)

Receiver Method: _on_button_pressed

Add Extra Call Argument: bool

Extra Call Arguments:

Unbind Signal Arguments: 0

☐ Deferred

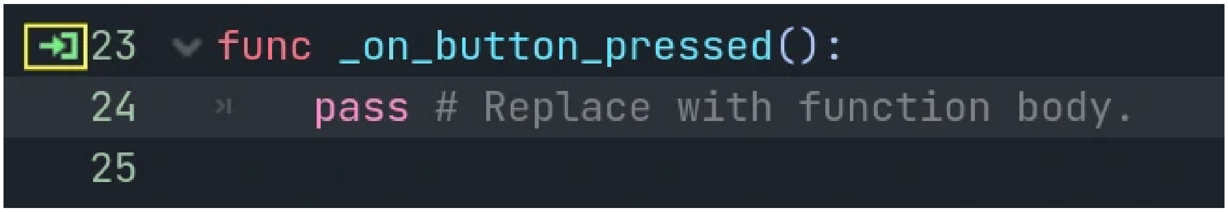
☐ Oneshot

The advanced view lets you connect to any node and any built-in function, add arguments to the callback, and set options. You can toggle the mode in the window's bottom-right by clicking the Advanced button.

Note

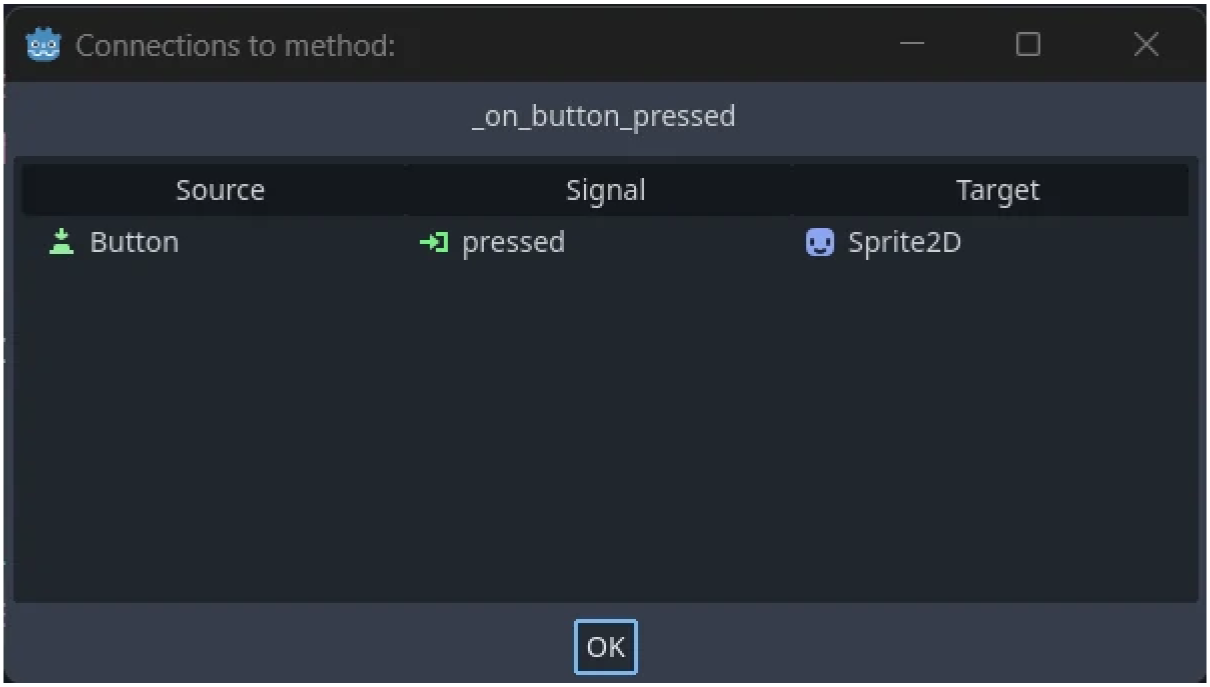
If you are using an external editor (such as VS Code) this automatic code generation might not work. In this case you need to to connect the signal via code as explained in the next section.

Click the Connect button to complete the signal connection and jump to the Script workspace. You should see the new method with a connection icon in the left margin.



```
func _on_button_pressed():
	pass # Replace with function body.

```

If you click the icon, a window pops up and displays information about the connection. This feature is only available when connecting nodes in the editor.



Source	Signal	Target
Button	pressed	Sprite2D

Let's replace the line with the `pass` keyword with code that'll toggle the node's motion.

Our Sprite2D moves thanks to code in the `_process()` function. Godot provides a method to toggle processing on and off: [Node.set_process()](). Another method of the Node class, `is_processing()`, returns `true` if idle processing is active. We can use the `not` keyword to invert the value.

GDScript

```
func _on_button_pressed():
    set_process(not is_processing())
```

C#

```
// We also specified this function name in PascalCase in the
editor's connection window.
```

```
private void OnButtonPressed()
{
    SetProcess(!IsProcessing());
}
```

This function will toggle processing and, in turn, the icon's motion on and off upon pressing the button.

Before trying the game, we need to simplify our `_process()` function to move the node automatically and not wait for user input. Replace it with the following code, which we saw two lessons ago:

GDScript

```
func _process(delta):
    rotation += angular_speed * delta
    var velocity = Vector2.UP.rotated(rotation) * speed
    position += velocity * delta
```

C#

```
public override void _Process(double delta)
{
    Rotation += _angularSpeed * (float)delta;
    var velocity = Vector2.Up.Rotated(Rotation) * _speed;
    Position += velocity * (float)delta;
}
```

Your complete `sprite_2d.gd` code should look like the following.

GDScript

```
extends Sprite2D

var speed = 400
var angular_speed = PI


func _process(delta):
    rotation += angular_speed * delta
    var velocity = Vector2.UP.rotated(rotation) * speed
    position += velocity * delta


func _on_button_pressed():
    set_process(not is_processing())
```

C#

```
using Godot;

public partial class MySprite2D : Sprite2D
{
    private float _speed = 400;
    private float _angularSpeed = Mathf.Pi;

    public override void _Process(double delta)
    {
        Rotation += _angularSpeed * (float)delta;
        var velocity = Vector2.Up.Rotated(Rotation) * _speed;
        Position += velocity * (float)delta;
    }

    // We also specified this function name in PascalCase in
the editor's connection window.
    private void OnButtonPressed()
    {
        SetProcess(!IsProcessing());
    }
}
```

Run the scene now and click the button to see the sprite start and stop.

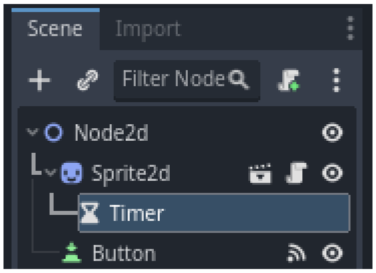
Connecting a signal via code

You can connect signals via code instead of using the editor. This is necessary when you create nodes or instantiate scenes inside of a script.

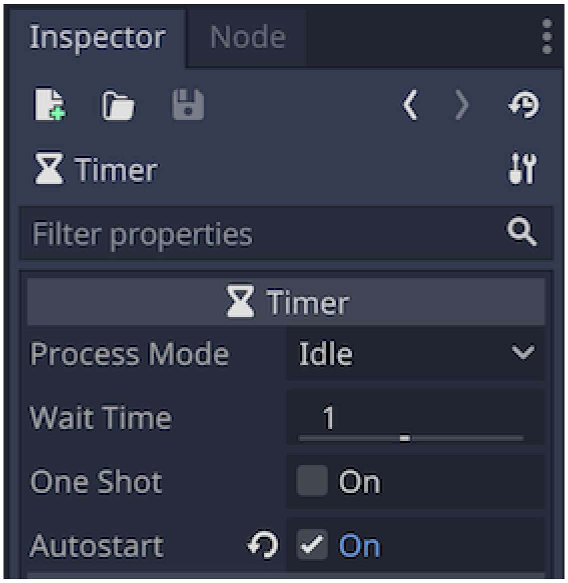
Let's use a different node here. Godot has a Timer node that's useful to implement skill cooldown times, weapon reloading, and more.

Head back to the 2D workspace. You can either click the "2D" text at the top of the window or press `Ctrl + F1` (`Ctrl + Cmd + 1` on macOS).

In the Scene dock, right-click on the Sprite2D node and add a new child node. Search for Timer and add the corresponding node. Your scene should now look like this.


Scene
Import
Filter Node
Node2d
Sprite2d
Timer
Button

With the Timer node selected, go to the Inspector and enable the **Autostart** property.


Inspector
Node
Timer
Filter properties
Timer

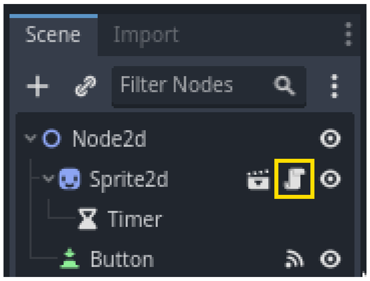
Process Mode Idle

Wait Time 1

One Shot ☐ On

Autostart ☑ On

Click the script icon next to Sprite2D to jump back to the scripting workspace.



We need to do two operations to connect the nodes via code:

1. Get a reference to the Timer from the Sprite2D.
2. Call the `connect()` method on the Timer's "timeout" signal.

Note

To connect to a signal via code, you need to call the `connect()` method of the signal you want to listen to. In this case, we want to listen to the Timer's "timeout" signal.

We want to connect the signal when the scene is instantiated, and we can do that using the Node._ready() built-in function, which is called automatically by the engine when a node is fully instantiated.

To get a reference to a node relative to the current one, we use the method Node.get_node(). We can store the reference in a variable.

GDScript

```
func _ready():
    var timer = get_node("Timer")
```

C#

```
public override void _Ready()
{
    var timer = GetNode<Timer>("Timer");
}
```

The function `get_node()` looks at the Sprite2D's children and gets nodes by their name. For example, if you renamed the Timer node to "BlinkingTimer" in the editor, you would have to change the call to `get_node("BlinkingTimer")`.

We can now connect the Timer to the Sprite2D in the `_ready()` function.

GDScript

```
func _ready():
    var timer = get_node("Timer")
    timer.timeout.connect(_on_timer_timeout)
```

C#

```
public override void _Ready()
{
    var timer = GetNode<Timer>("Timer");
    timer.Timeout += OnTimerTimeout;
}
```

The line reads like so: we connect the Timer's "timeout" signal to the node to which the script is attached. When the Timer emits `timeout`, we want to call the function `_on_timer_timeout()`, that we need to define. Let's add it at the bottom of our script and use it to toggle our sprite's visibility.

Note

By convention, we name these callback methods in GDScript as "_on_node_name_signal_name" and in C# as "OnNodeNameSignalName". Here, it'll be "_on_timer_timeout" for GDScript and OnTimerTimeout() for C#.

GDScript

```
func _on_timer_timeout():
    visible = not visible
```

C#

```
private void OnTimerTimeout()
{
    Visible = !Visible;
}
```

The `visible` property is a boolean that controls the visibility of our node. The line `visible = not visible` toggles the value. If `visible` is `true`, it becomes `false`, and vice-versa.

If you run the Node2D scene now, you will see that the sprite blinks on and off, at one second intervals.

Complete script

That's it for our little moving and blinking Godot icon demo! Here is the complete `sprite_2d.gd` file for reference.

GDScript

```
extends Sprite2D

var speed = 400
var angular_speed = PI


func _ready():
    var timer = get_node("Timer")
    timer.timeout.connect(_on_timer_timeout)


func _process(delta):
    rotation += angular_speed * delta
    var velocity = Vector2.UP.rotated(rotation) * speed
    position += velocity * delta
```

```
func _on_button_pressed():
    set_process(not is_processing())


func _on_timer_timeout():
    visible = not visible
```

C#

```
using Godot;

public partial class MySprite2D : Sprite2D
{
    private float _speed = 400;
    private float _angularSpeed = Mathf.Pi;

    public override void _Ready()
    {
        var timer = GetNode<Timer>("Timer");
        timer.Timeout += OnTimerTimeout;
    }

    public override void _Process(double delta)
    {
        Rotation += _angularSpeed * (float)delta;
        var velocity = Vector2.Up.Rotated(Rotation) * _speed;
        Position += velocity * (float)delta;
    }

    // We also specified this function name in PascalCase in
the editor's connection window.
    private void OnButtonPressed()
    {
        SetProcess(!IsProcessing());
    }

    private void OnTimerTimeout()
    {
        Visible = !Visible;
    }
}
```

Custom signals

Note

This section is a reference on how to define and use your own signals, and does not build upon the project created in previous lessons.

You can define custom signals in a script. Say, for example, that you want to show a game over screen when the player's health reaches zero. To do so, you could define a signal named "died" or "health_depleted" when their health reaches 0.

GDScript

```
extends Node2D

signal health_depleted

var health = 10
```

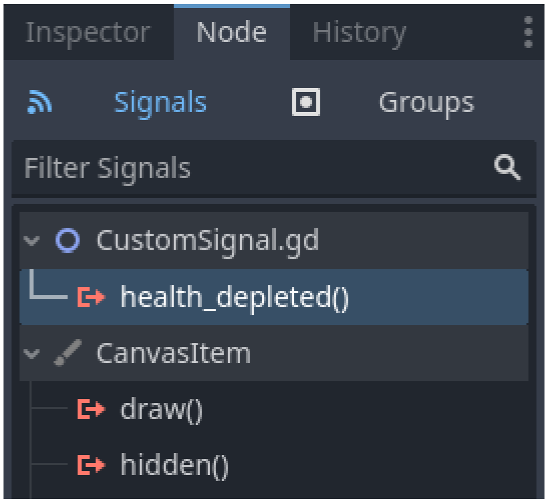
C#

```
using Godot;

public partial class MyNode2D : Node2D
{
    [Signal]
    public delegate void HealthDepletedEventHandler();

    private int _health = 10;
}
```

Note

As signals represent events that just occurred, we generally use an action verb in the past tense in their names.

Your signals work the same way as built-in ones: they appear in the Node tab and you can connect to them like any other.


Inspector
Node
History
Signals
Groups
Filter Signals
CustomSignal.gd
health_depleted()
CanvasItem
draw()
hidden()

To emit a signal in your scripts, call `emit()` on the signal.

GDScript

```
func take_damage(amount):
    health -= amount
    if health <= 0:
        health_depleted.emit()
```

C#

```
public void TakeDamage(int amount)
{
    _health -= amount;

    if (_health <= 0)
    {
        EmitSignal(SignalName.HealthDepleted);
    }
}
```

A signal can optionally declare one or more arguments. Specify the argument names between parentheses:

GDScript

```
extends Node2D

signal health_changed(old_value, new_value)

var health = 10
```

C#

```
using Godot;

public partial class MyNode : Node
{
    [Signal]
    public delegate void HealthChangedEventHandler(int
oldValue, int newValue);

    private int _health = 10;
}
```

Note

The signal arguments show up in the editor's node dock, and Godot can use them to generate callback functions for you. However, you can still emit any number of arguments when you emit signals. So it's up to you to emit the correct values.

To emit values along with the signal, add them as extra arguments to the `emit()` function:

GDScript

```
func take_damage(amount):
    var old_health = health
    health -= amount
    health_changed.emit(old_health, health)
```

C#

```
public void TakeDamage(int amount)
{
    int oldHealth = _health;
    _health -= amount;
    EmitSignal(SignalName.HealthChanged, oldHealth, _health);
}
```

Summary

Any node in Godot emits signals when something specific happens to them, like a button being pressed. Other nodes can connect to individual signals and react to selected events.

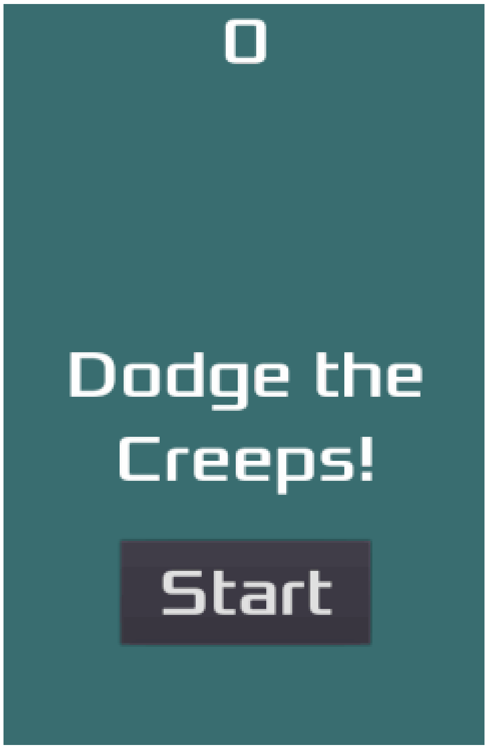
Signals have many uses. With them, you can react to a node entering or exiting the game world, to a collision, to a character entering or leaving an area, to an element of the interface changing size, and much more.

For example, an Area2D representing a coin emits a `body_entered` signal whenever the player's physics body enters its collision shape, allowing you to know when the player collected it.

In the next section, Your first 2D game, you'll create a complete 2D game and put everything you learned so far into practice.

Your first 2D game

In this step-by-step tutorial series, you will create your first complete 2D game with Godot. By the end of the series, you will have a simple yet complete game of your own, like the image below.



You will learn how the Godot editor works, how to structure a project, and build a 2D game.

Note

This project is an introduction to the Godot engine. It assumes that you have some programming experience already. If you're new to programming entirely, you should start here: Scripting languages.

The game is called "Dodge the Creeps!". Your character must move and avoid the enemies for as long as possible.

You will learn to:

- Create a complete 2D game with the Godot editor.
- Structure a simple game project.
- Move the player character and change its sprite.
- Spawn random enemies.
- Count the score.

And more.

You'll find another series where you'll create a similar game but in 3D. We recommend you to start with this one, though.

Why start with 2D?

If you are new to game development or unfamiliar with Godot, we recommend starting with 2D games. This will allow you to become comfortable with both before tackling 3D games, which tend to be more complicated.

You can find a completed version of this project at this location:

- https://github.com/godotengine/godot-demo-projects/tree/master/2d/dodge_the_creeps

Prerequisites

This step-by-step tutorial is intended for beginners who followed the complete Getting Started.

If you're an experienced programmer, you can find the complete demo's source code here: Dodge the Creeps source code [https://github.com/godotengine/godot-demo-projects/tree/master/2d/dodge_the_creeps].

We prepared some game assets you'll need to download so we can jump straight to the code.

You can download them by clicking the link below.

dodge_the_creeps_2d_assets.zip [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/dodge_the_creeps_2d_assets.zip].

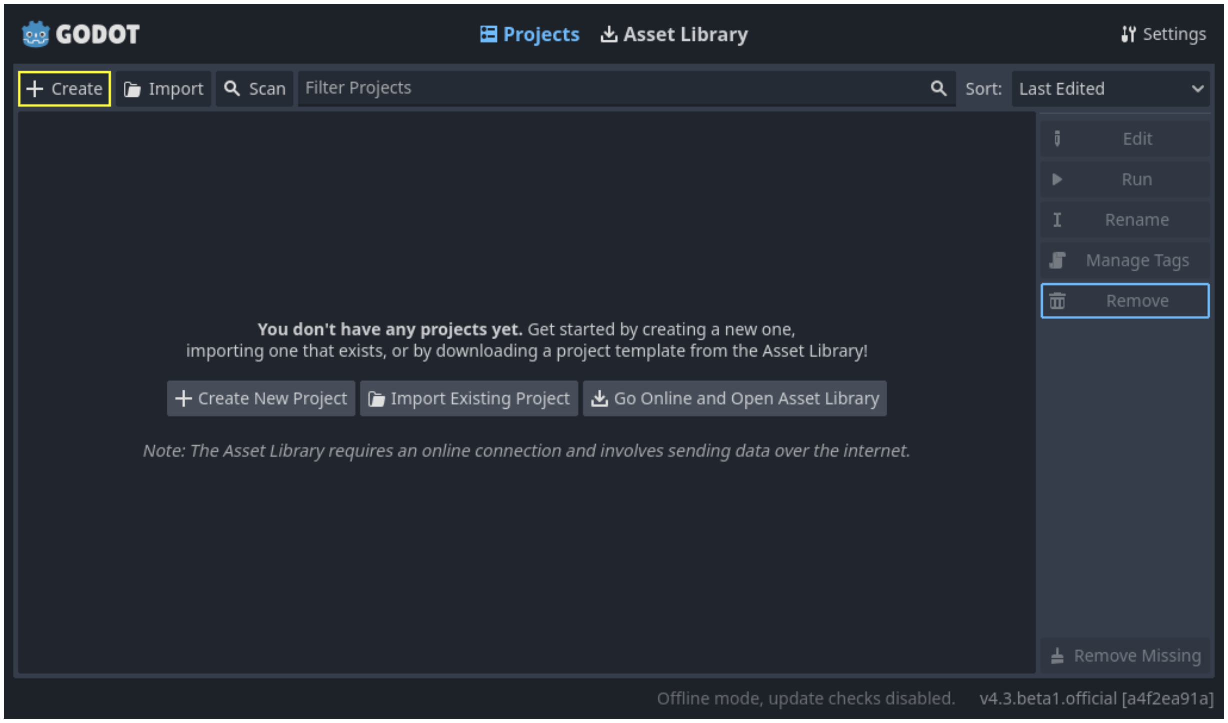
Contents

- Setting up the project
- Creating the player scene
- Coding the player
- Creating the enemy
- The main game scene
- Heads up display
- Finishing up

Setting up the project

In this short first part, we'll set up and organize the project.

Launch Godot and create a new project.


GODOT
Projects
Asset Library
Settings
Create
Import
Scan
Filter Projects
Sort:
Last Edited
Edit
Run
Rename
Manage Tags
Remove
Create New Project
Import Existing Project
Go Online and Open Asset Library
Remove Missing

You don't have any projects yet. Get started by creating a new one, importing one that exists, or by downloading a project template from the Asset Library!

Note: The Asset Library requires an online connection and involves sending data over the internet.

Offline mode, update checks disabled. v4.3.beta1.official [a4f2ea91a]

When creating the new project, you only need to choose a valid *Project Path*. You can leave the other default settings alone.

GDScript

Download [dodge_the_creeps_2d_assets.zip](https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/dodge_the_creeps_2d_assets.zip) [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/dodge_the_creeps_2d_assets.zip]. The archive contains the images and sounds you'll be using to make the game. Extract the archive and move the `art/` and `fonts/` directories to your project's directory.

C#

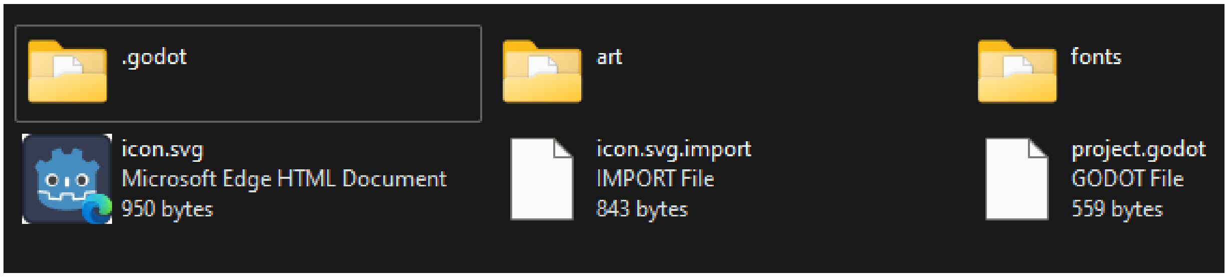
Download dodge_the_creeps_2d_assets.zip [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/dodge_the_creeps_2d_assets.zip]. The archive contains the images and sounds you'll be using to make the game. Extract the archive and move the `art/` and `fonts/` directories to your project's directory.

Ensure that you have the required dependencies to use C# in Godot. You need the latest stable .NET SDK, and an editor such as VS Code. See Prerequisites.

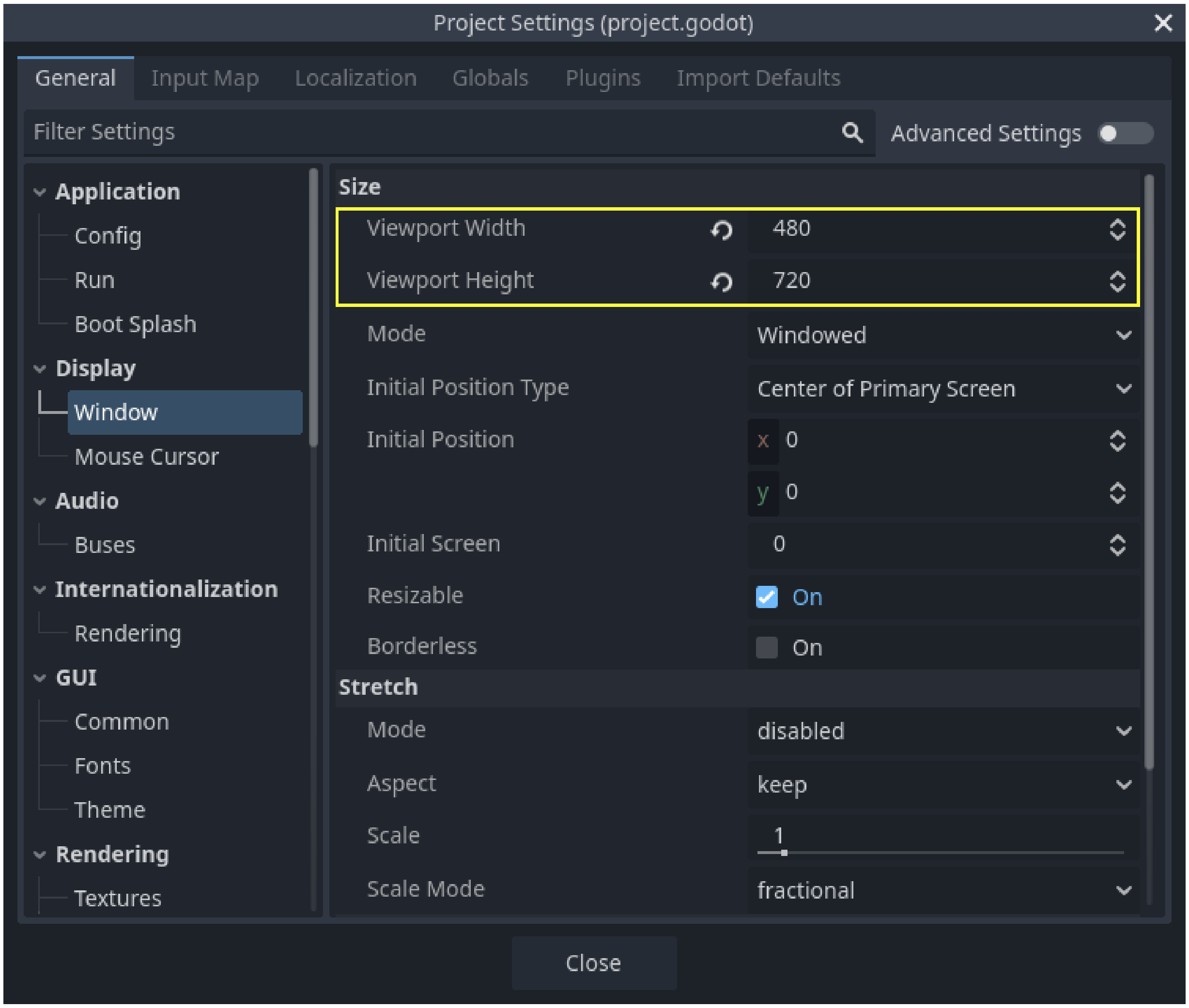
C++

The C++ part of this tutorial wasn't rewritten for the new GDExtension system yet.

Your project folder should look like this.


.godot
art
fonts
icon.svg
Microsoft Edge HTML Document
950 bytes
icon.svg.import
IMPORT File
843 bytes
project.godot
GODOT File
559 bytes

This game is designed for portrait mode, so we need to adjust the size of the game window. Click on *Project -> Project Settings* to open the project settings window, in the left column open the *Display -> Window* tab. There, set "Viewport Width" to `480` and "Viewport Height" to `720`.


Project Settings (project.godot)
General
Input Map
Localization
Globals
Plugins
Import Defaults
Filter Settings
Advanced Settings
Application
Config
Run
Boot Splash
Display
Window
Mouse Cursor
Audio
Buses
Internationalization
Rendering
GUI
Common
Fonts
Theme
Rendering
Textures
Close

Size

Viewport Width 480

Viewport Height 720

Mode Windowed

Initial Position Type Center of Primary Screen

Initial Position x 0 y 0

Initial Screen 0

Resizable ☑ On

Borderless ☐ On

Stretch

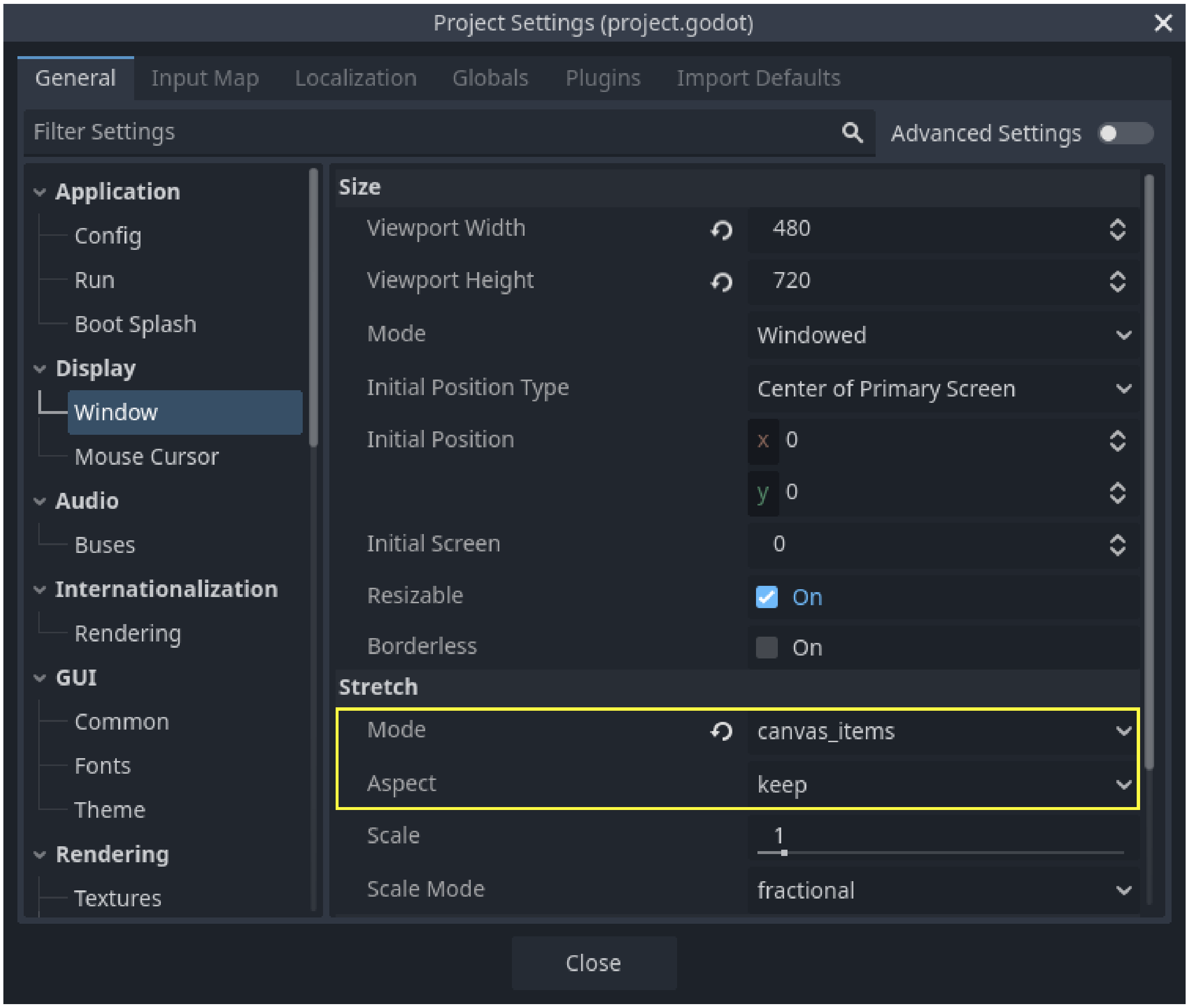
Mode disabled

Aspect keep

Scale 1

Scale Mode fractional

Also, under the **Stretch** options, set **Mode** to `canvas_items` and **Aspect** to `keep`. This ensures that the game scales consistently on different sized screens.


Project Settings (project.godot)
General
Input Map
Localization
Globals
Plugins
Import Defaults
Filter Settings
Advanced Settings
Application
Config
Run
Boot Splash
Display
Window
Mouse Cursor
Audio
Buses
Internationalization
Rendering
GUI
Common
Fonts
Theme
Rendering
Textures
Close

Size

Viewport Width: 480

Viewport Height: 720

Mode: Windowed

Initial Position Type: Center of Primary Screen

Initial Position: x 0, y 0

Initial Screen: 0

Resizable: ☑ On

Borderless: ☐ On

Stretch

Mode: canvas_items

Aspect: keep

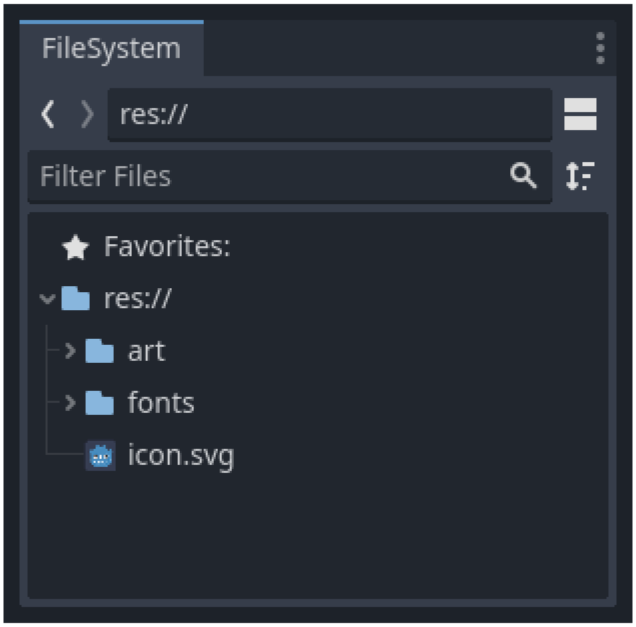
Scale: 1

Scale Mode: fractional

Organizing the project

In this project, we will make 3 independent scenes: `Player`, `Mob`, and `HUD`, which we will combine into the game's `Main` scene.

In a larger project, it might be useful to create folders to hold the various scenes and their scripts, but for this relatively small game, you can save your scenes and scripts in the project's root folder, identified by `res://`. You can see your project folders in the FileSystem dock in the lower left corner:


FileSystem
res://
Filter Files
Favorites:
res://
art
fonts
icon.svg

With the project in place, we're ready to design the player scene in the next lesson.

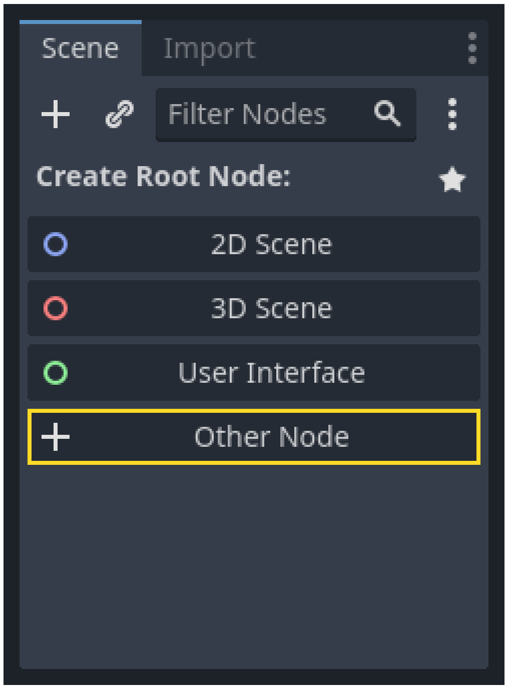
Creating the player scene

With the project settings in place, we can start working on the player-controlled character.

The first scene will define the `Player` object. One of the benefits of creating a separate Player scene is that we can test it separately, even before we've created other parts of the game.

Node structure

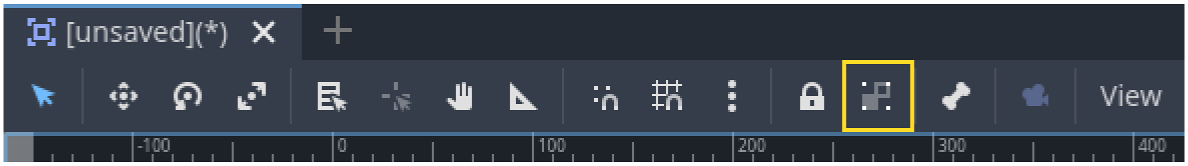
To begin, we need to choose a root node for the player object. As a general rule, a scene's root node should reflect the object's desired functionality - what the object *is*. Click the "Other Node" button and add an Area2D node to the scene.


Scene
Import
Filter Nodes
Create Root Node:
2D Scene
3D Scene
User Interface
Other Node

Godot will display a warning icon next to the node in the scene tree. You can ignore it for now. We will address it later.

With `Area2D` we can detect objects that overlap or run into the player. Change the node's name to `Player` by double-clicking on it. Now that we've set the scene's root node, we can add additional nodes to give it more functionality.

Before we add any children to the `Player` node, we want to make sure we don't accidentally move or resize them by clicking on them. Select the node and click the icon to the right of the lock. Its tooltip says "Groups the selected node with its children. This causes the parent to be selected when any child node is clicked in 2D and 3D view."



Save the scene. Click Scene -> Save, or press `Ctrl + S` on Windows/Linux or `Cmd + S` on macOS.

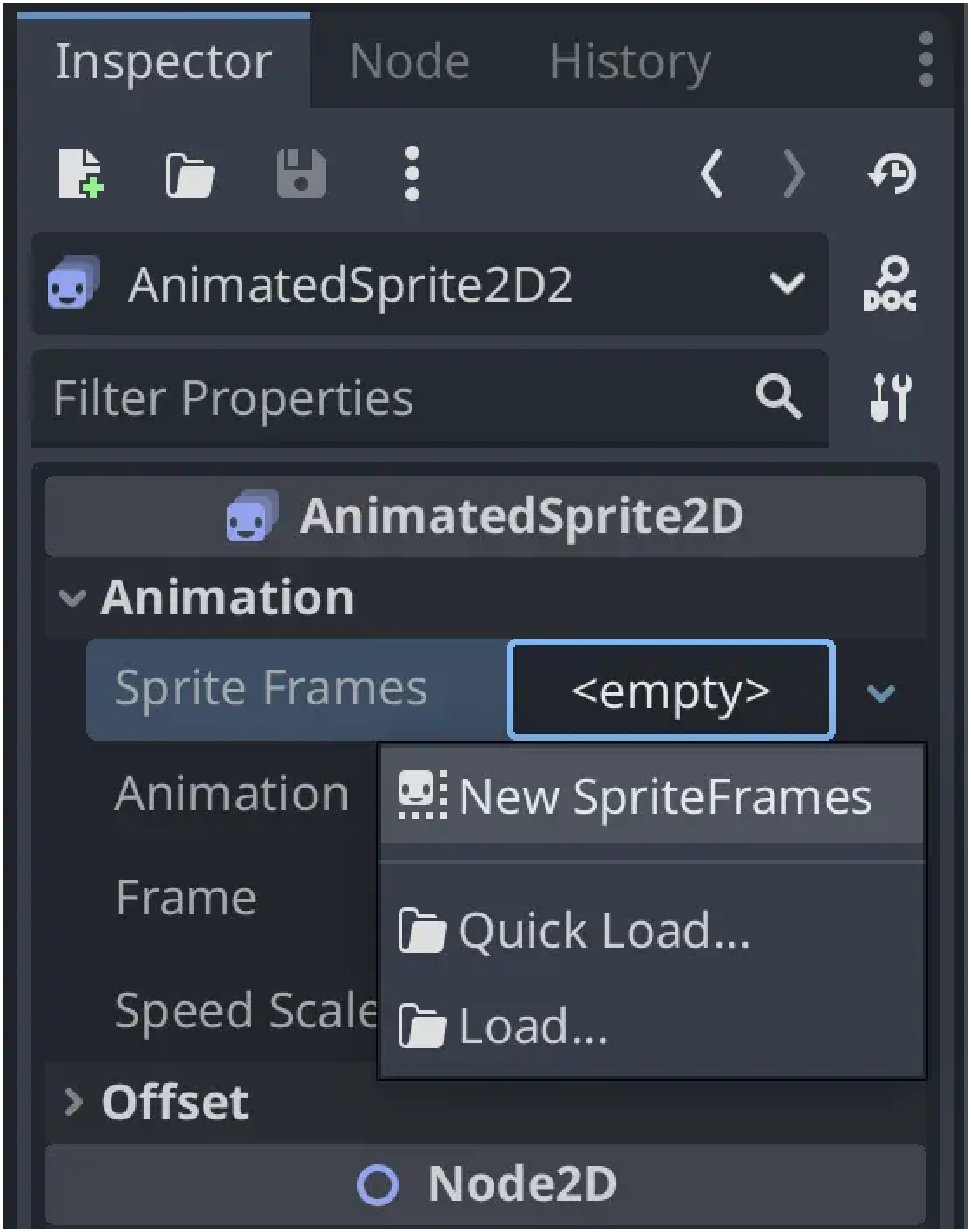
Note

For this project, we will be following the Godot naming conventions.

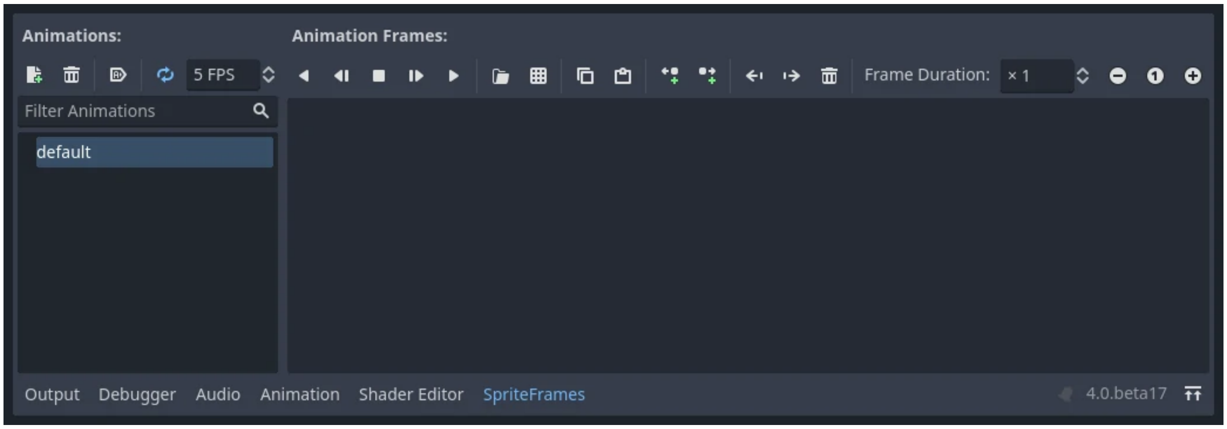
- **GDScript**: Classes (nodes) use PascalCase, variables and functions use snake_case, and constants use ALL_CAPS (See GDScript style guide).
- **C#**: Classes, export variables and methods use PascalCase, private fields use _camelCase, local variables and parameters use camelCase (See C# style guide). Be careful to type the method names precisely when connecting signals.

Sprite animation

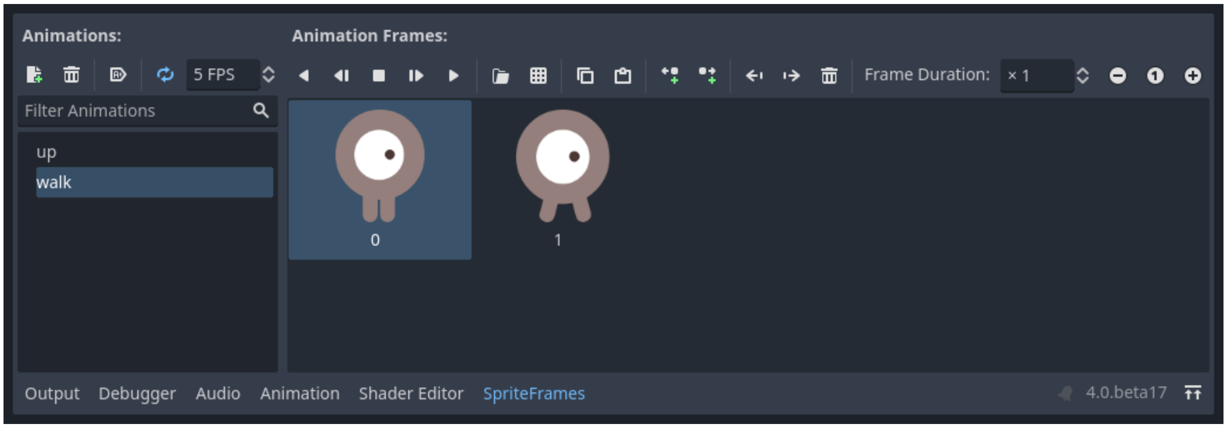
Click on the `Player` node and add (`Ctrl + A` on Windows/Linux or `Cmd + A` on macOS) a child node AnimatedSprite2D. The `AnimatedSprite2D` will handle the appearance and animations for our player. Notice that there is a warning symbol next to the node. An `AnimatedSprite2D` requires a SpriteFrames resource, which is a list of the animations it can display. To create one, find the `Sprite Frames` property under the `Animation` tab in the Inspector and click "[empty]" -> "New SpriteFrames":


Inspector
Node
History
AnimatedSprite2D2
Filter Properties
AnimatedSprite2D
Animation
Sprite Frames
<empty>
Animation
Frame
Speed Scale
New SpriteFrames
Quick Load...
Load...
Offset
Node2D

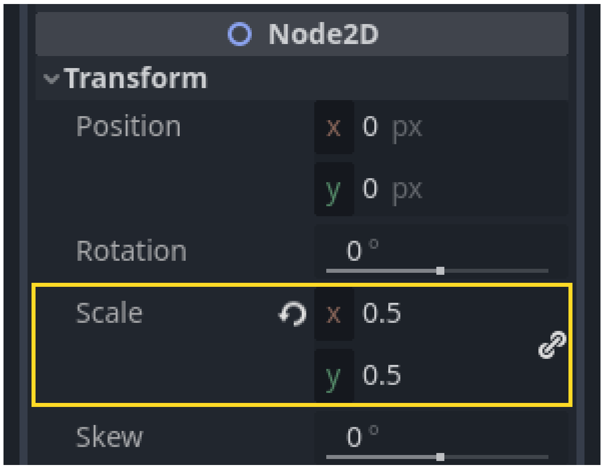
Click on the `SpriteFrames` you just created to open the "SpriteFrames" panel:


Animations:
Animation Frames:
5 FPS
Frame Duration:
× 1
Filter Animations
default
Output
Debugger
Audio
Animation
Shader Editor
SpriteFrames
4.0.beta17

On the left is a list of animations. Click the "default" one and rename it to "walk". Then click the "Add Animation" button to create a second animation named "up". Find the player images in the "FileSystem" tab - they're in the `art` folder you unzipped earlier. Drag the two images for each animation, named `playerGrey_walk[1/2]` and `playerGrey_walk[2/2]`, into the "Animation Frames" side of the panel for the corresponding animation:


Animations:
Animation Frames:
5 FPS
Frame Duration:
× 1
Filter Animations
up
walk
0
1
Output
Debugger
Audio
Animation
Shader Editor
SpriteFrames
4.0.beta17

The player images are a bit too large for the game window, so we need to scale them down. Click on the `AnimatedSprite2D` node and set the `Scale` property to `(0.5, 0.5)`. You can find it in the Inspector under the `Node2D` heading.



Node2D

Transform

Position x 0 px

y 0 px

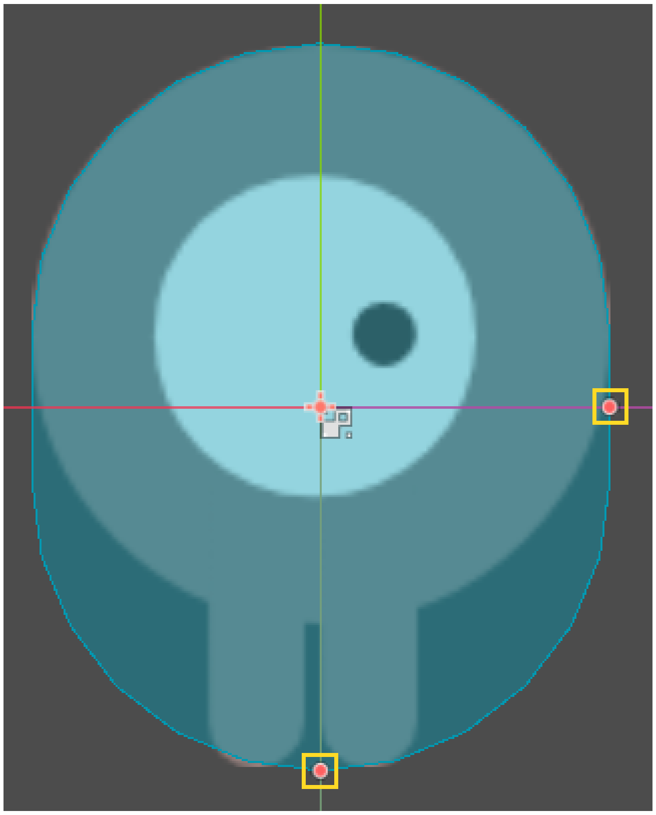
Rotation 0 °

Scale x 0.5

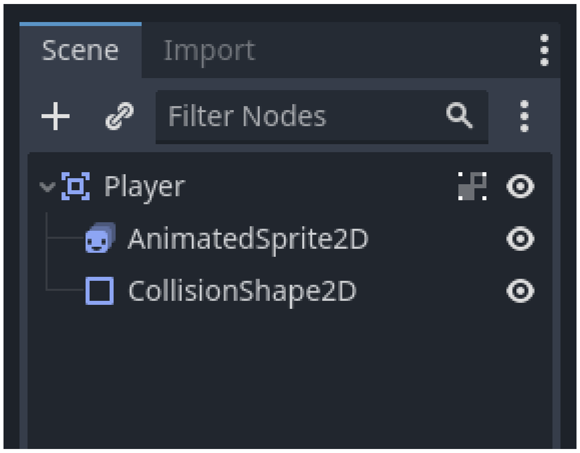
y 0.5

Skew 0 °

Finally, add a CollisionShape2D as a child of `Player`. This will determine the player's "hitbox", or the bounds of its collision area. For this character, a `CapsuleShape2D` node gives the best fit, so next to "Shape" in the Inspector, click "[empty]" -> "New CapsuleShape2D". Using the two size handles, resize the shape to cover the sprite:



When you're finished, your `Player` scene should look like this:


Scene
Import
Filter Nodes
Player
AnimatedSprite2D
CollisionShape2D

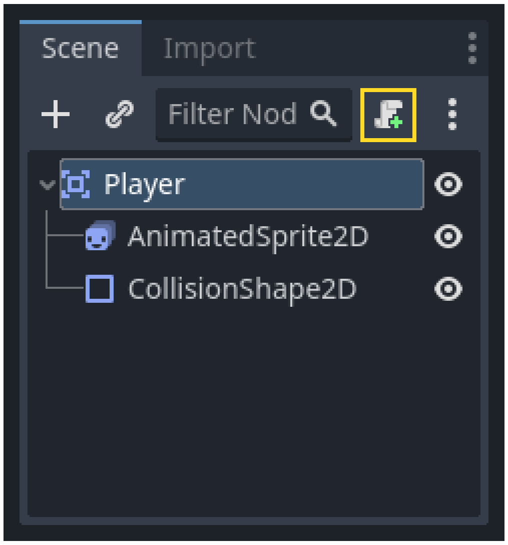
Make sure to save the scene again after these changes.

In the next part, we'll add a script to the player node to move and animate it. Then, we'll set up collision detection to know when the player got hit by something.

Coding the player

In this lesson, we'll add player movement, animation, and set it up to detect collisions.

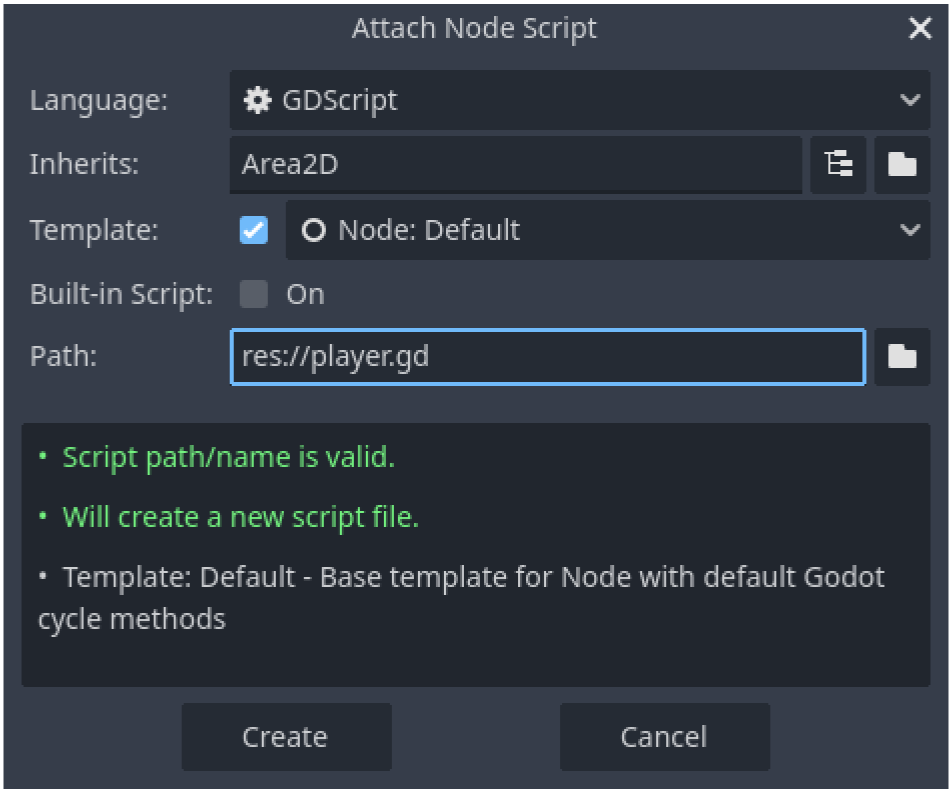
To do so, we need to add some functionality that we can't get from a built-in node, so we'll add a script. Click the `Player` node and click the "Attach Script" button:


Scene
Import
Filter Nod
Player
AnimatedSprite2D
CollisionShape2D

In the script settings window, you can leave the default settings alone. Just click "Create":

Note

If you're creating a C# script or other languages, select the language from the *language* drop down menu before hitting create.



Language: GDScript

Inherits: Area2D

Template: ☑ Node: Default

Built-in Script: ☐ On

Path: res://player.gd

- Script path/name is valid.
- Will create a new script file.
- Template: Default - Base template for Node with default Godot cycle methods

Note

If this is your first time encountering GDScript, please read [Scripting languages](#) before continuing.

Start by declaring the member variables this object will need:

GDScript

```
extends Area2D

@export var speed = 400 # How fast the player will move (pixels/sec).
var screen_size # Size of the game window.
```

C#

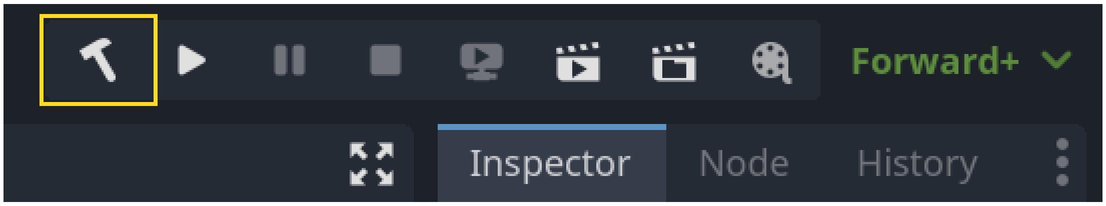
```
using Godot;

public partial class Player : Area2D
```

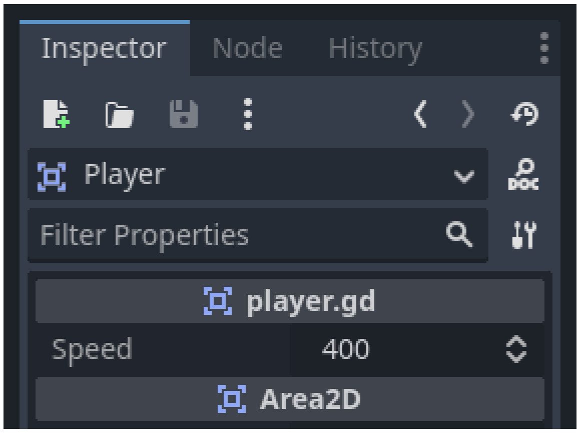
```
{
    [Export]
    public int Speed { get; set; } = 400; // How fast the
player will move (pixels/sec).

    public Vector2 ScreenSize; // Size of the game window.
}
```

Using the `export` keyword on the first variable `speed` allows us to set its value in the Inspector. This can be handy for values that you want to be able to adjust just like a node's built-in properties. Click on the `Player` node and you'll see the property now appears in the Inspector in a new section with the name of the script. Remember, if you change the value here, it will override the value written in the script.

Warning

If you're using C#, you need to (re)build the project assemblies whenever you want to see new export variables or signals. This build can be manually triggered by clicking the **Build** button at the top right of the editor.





player.gd
Speed 400
Area2D

Your `player.gd` script should already contain a `_ready()` and a `_process()` function. If you didn't select the default template shown above, create these functions while following the lesson.

The `_ready()` function is called when a node enters the scene tree, which is a good time to find the size of the game window:

GDScript

```
func _ready():
    screen_size = get_viewport_rect().size
```

C#

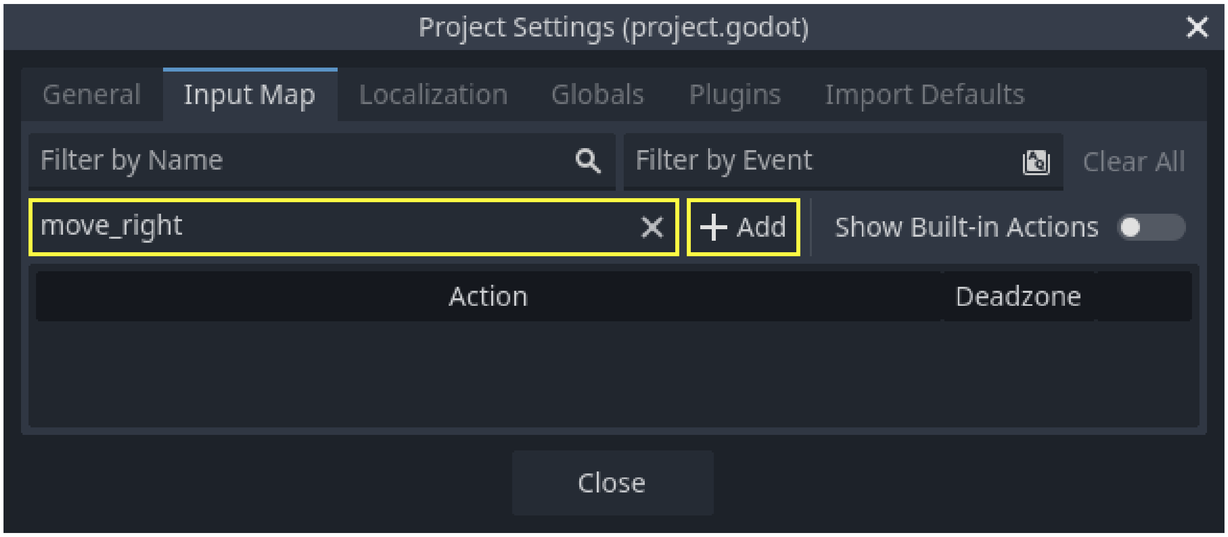
```
public override void _Ready()
{
    ScreenSize = GetViewportRect().Size;
}
```

Now we can use the `_process()` function to define what the player will do. `_process()` is called every frame, so we'll use it to update elements of our game, which we expect will change often. For the player, we need to do the following:

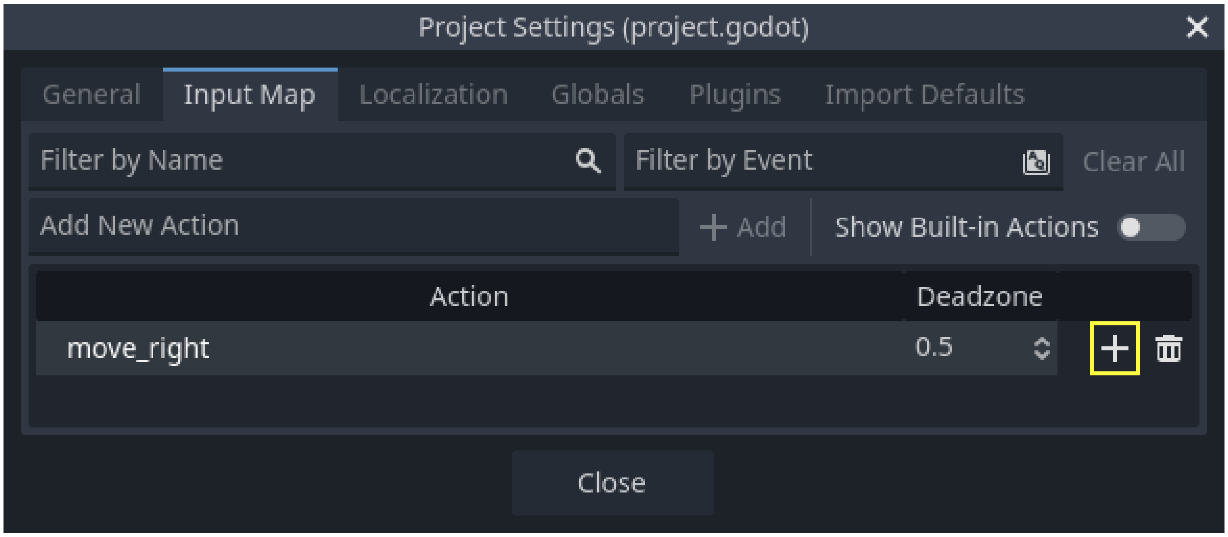
- Check for input.
- Move in the given direction.
- Play the appropriate animation.

First, we need to check for input - is the player pressing a key? For this game, we have 4 direction inputs to check. Input actions are defined in the Project Settings under "Input Map". Here, you can define custom events and assign different keys, mouse events, or other inputs to them. For this game, we will map the arrow keys to the four directions.

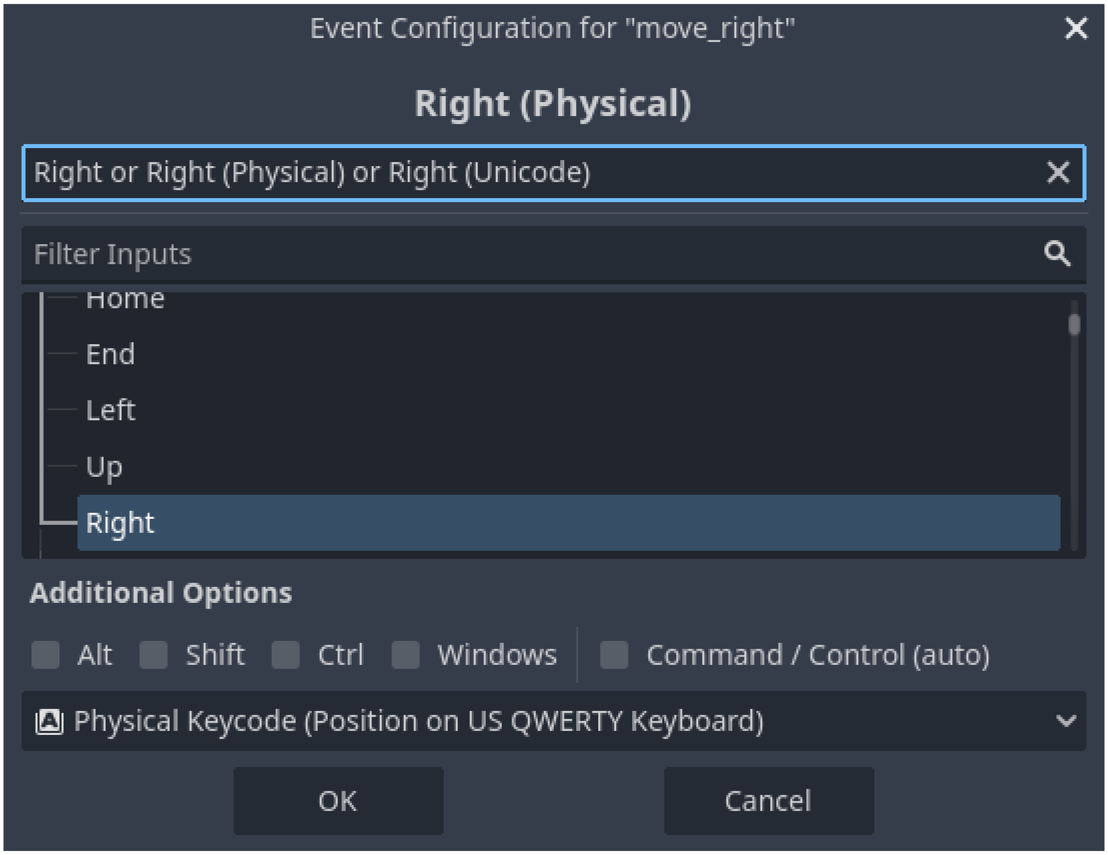
Click on *Project -> Project Settings* to open the project settings window and click on the *Input Map* tab at the top. Type "move_right" in the top bar and click the "Add" button to add the `move_right` action.


Project Settings (project.godot)
General
Input Map
Localization
Globals
Plugins
Import Defaults
Filter by Name
Filter by Event
Clear All
move_right
+ Add
Show Built-in Actions
Action
Deadzone
Close

We need to assign a key to this action. Click the "+" icon on the right, to open the event manager window.


Project Settings (project.godot)
General
Input Map
Localization
Globals
Plugins
Import Defaults
Filter by Name
Filter by Event
Clear All
Add New Action
+ Add
Show Built-in Actions
Action
Deadzone
move_right
0.5
Close

The "Listening for Input..." field should automatically be selected. Press the "right" key on your keyboard, and the menu should look like this now.

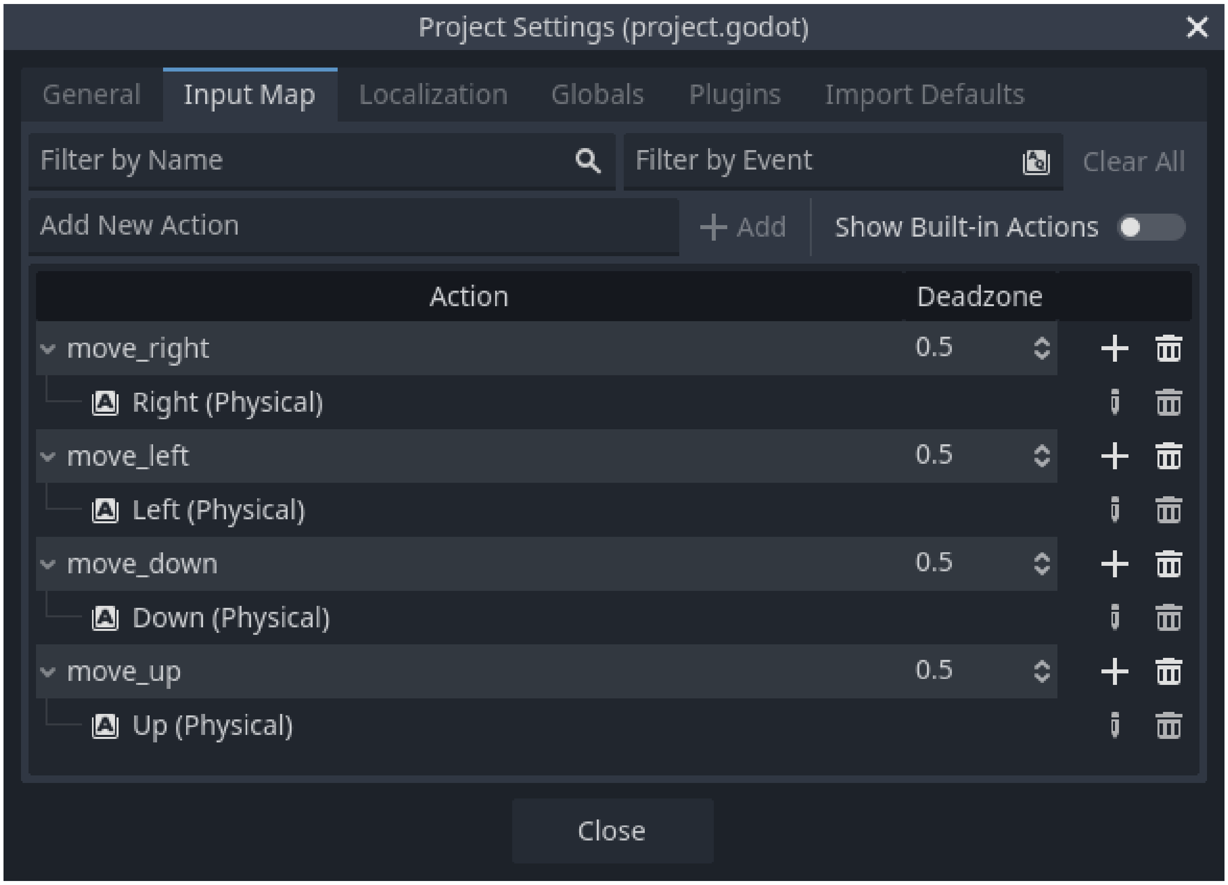

Event Configuration for "move_right"
Right (Physical)
Right or Right (Physical) or Right (Unicode)
Filter Inputs
Home
End
Left
Up
Right
Additional Options
Alt
Shift
Ctrl
Windows
Command / Control (auto)
Physical Keycode (Position on US QWERTY Keyboard)
OK
Cancel

Select the "ok" button. The "right" key is now associated with the `move_right` action.

Repeat these steps to add three more mappings:

1. `move_left` mapped to the left arrow key.
2. `move_up` mapped to the up arrow key.
3. And `move_down` mapped to the down arrow key.

Your input map tab should look like this:


Project Settings (project.godot)
General
Input Map
Localization
Globals
Plugins
Import Defaults
Filter by Name
Filter by Event
Clear All
Add New Action
Add
Show Built-in Actions
Close

Action	Deadzone
move_right	0.5
Right (Physical)	
move_left	0.5
Left (Physical)	
move_down	0.5
Down (Physical)	
move_up	0.5
Up (Physical)	

Click the "Close" button to close the project settings.

Note

We only mapped one key to each input action, but you can map multiple keys, joystick buttons, or mouse buttons to the same input action.

You can detect whether a key is pressed using `Input.is_action_pressed()`, which returns `true` if it's pressed or `false` if it isn't.

GDScript

```
func _process(delta):
    var velocity = Vector2.ZERO # The player's movement vector.
    if Input.is_action_pressed("move_right"):
        velocity.x += 1
    if Input.is_action_pressed("move_left"):
        velocity.x -= 1
    if Input.is_action_pressed("move_down"):
        velocity.y += 1
    if Input.is_action_pressed("move_up"):
        velocity.y -= 1

    if velocity.length() > 0:
        velocity = velocity.normalized() * speed
        $AnimatedSprite2D.play()
    else:
        $AnimatedSprite2D.stop()
```

C#

```
public override void _Process(double delta)
{
    var velocity = Vector2.Zero; // The player's movement vector.

    if (Input.IsActionPressed("move_right"))
    {
        velocity.X += 1;
    }

    if (Input.IsActionPressed("move_left"))
    {
        velocity.X -= 1;
    }

    if (Input.IsActionPressed("move_down"))
    {
        velocity.Y += 1;
    }

    if (Input.IsActionPressed("move_up"))
    {
        velocity.Y -= 1;
    }

    var animatedSprite2D = GetNode<AnimatedSprite2D>
("AnimatedSprite2D");

    if (velocity.Length() > 0)
    {
        velocity = velocity.Normalized() * Speed;
        animatedSprite2D.Play();
    }
    else
    {
        animatedSprite2D.Stop();
    }
}
```

We start by setting the `velocity` to `(0, 0)` - by default, the player should not be moving. Then we check each input and add/subtract from the `velocity` to obtain a total direction. For example, if you hold `right` and `down` at the same time, the resulting `velocity` vector will be `(1, 1)`. In this case, since we're adding a horizontal and a vertical movement, the player would move *faster* diagonally than if it just moved horizontally.

We can prevent that if we *normalize* the velocity, which means we set its *length* to `1`, then multiply by the desired speed. This means no more fast diagonal movement.

Tip

If you've never used vector math before, or need a refresher, you can see an explanation of vector usage in Godot at Vector math. It's good to know but won't be necessary for the rest of this tutorial.

We also check whether the player is moving so we can call `play()` or `stop()` on the AnimatedSprite2D.

Tip

`$` is shorthand for `get_node()`. So in the code above, `$AnimatedSprite2D.play()` is the same as `get_node("AnimatedSprite2D").play()`.

In GDScript, `$` returns the node at the relative path from the current node, or returns `null` if the node is not found. Since AnimatedSprite2D is a child of the current node, we can use `$AnimatedSprite2D`.

Now that we have a movement direction, we can update the player's position. We can also use `clamp()` to prevent it from leaving the screen. *Clamping* a value means restricting it to a given range. Add the following to the bottom of the `_process` function (make sure it's not indented under the *else*):

GDScript

```
position += velocity * delta
position = position.clamp(Vector2.ZERO, screen_size)
```

C#

```
Position += velocity * (float)delta;
Position = new Vector2(
    x: Mathf.Clamp(Position.X, 0, ScreenSize.X),
    y: Mathf.Clamp(Position.Y, 0, ScreenSize.Y)
);
```

Tip

The *delta* parameter in the *_process()* function refers to the *frame length* - the amount of time that the previous frame took to complete. Using this value ensures that your movement will remain consistent even if the frame rate changes.

Click "Run Current Scene" (`F6`, `Cmd + R` on macOS) and confirm you can move the player around the screen in all directions.

Warning

If you get an error in the "Debugger" panel that says

```
Attempt to call function 'play' in base 'null instance' on a null instance
```

this likely means you spelled the name of the AnimatedSprite2D node wrong. Node names are case-sensitive and `$NodeName` must match the name you see in the scene tree.

Choosing animations

Now that the player can move, we need to change which animation the AnimatedSprite2D is playing based on its direction. We have the "walk" animation, which shows the player walking to the right. This animation should be flipped horizontally using the `flip_h` property for left movement. We also have the "up" animation, which should be flipped vertically with `flip_v` for downward movement. Let's place this code at the end of the `_process()` function:

GDScript

```
if velocity.x != 0:
    $AnimatedSprite2D.animation = "walk"
    $AnimatedSprite2D.flip_v = false
    # See the note below about the following boolean
assignment.
    $AnimatedSprite2D.flip_h = velocity.x < 0
elif velocity.y != 0:
    $AnimatedSprite2D.animation = "up"
    $AnimatedSprite2D.flip_v = velocity.y > 0
```

C#

```
if (velocity.X != 0)
{
    animatedSprite2D.Animation = "walk";
    animatedSprite2D.FlipV = false;
    // See the note below about the following boolean
assignment.
    animatedSprite2D.FlipH = velocity.X < 0;
}
else if (velocity.Y != 0)
{
    animatedSprite2D.Animation = "up";
    animatedSprite2D.FlipV = velocity.Y > 0;
}
```

Note

The boolean assignments in the code above are a common shorthand for programmers. Since we're doing a comparison test (boolean) and also *assigning* a boolean value, we can do both at the same time. Consider this code versus the one-line boolean assignment above:

GDScript

```
if velocity.x < 0:
    $AnimatedSprite2D.flip_h = true
else:
    $AnimatedSprite2D.flip_h = false
```

C#

```
if (velocity.X < 0)
{
    animatedSprite2D.FlipH = true;
}
else
{
    animatedSprite2D.FlipH = false;
}
```

Play the scene again and check that the animations are correct in each of the directions.

Tip

A common mistake here is to type the names of the animations wrong. The animation names in the SpriteFrames panel must match what you type in the code. If you named the animation `"Walk"`, you must also use a capital "W" in the code.

When you're sure the movement is working correctly, add this line to `_ready()`, so the player will be hidden when the game starts:

GDScript

```
hide()
```

C#

```
Hide();
```

Preparing for collisions

We want `Player` to detect when it's hit by an enemy, but we haven't made any enemies yet! That's OK, because we're going to use Godot's *signal* functionality to make it work.

Add the following at the top of the script. If you're using GDScript, add it after `extends Area2D`. If you're using C#, add it after `public partial class Player : Area2D`:

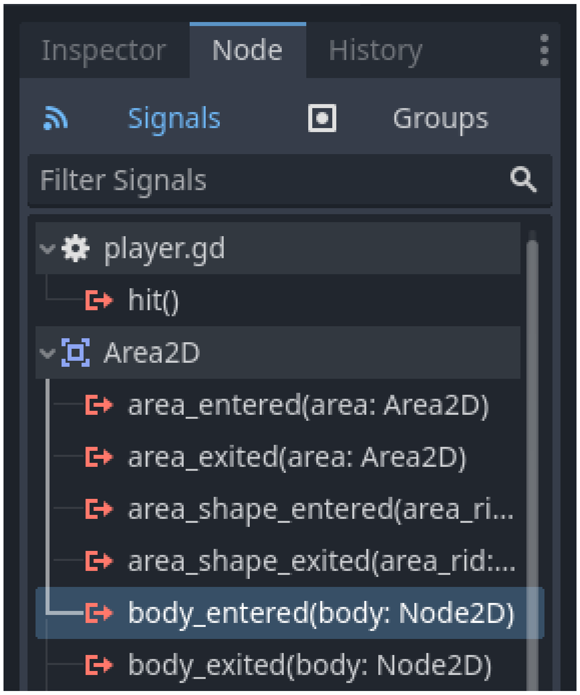
GDScript

```
signal hit
```

C#

```
// Don't forget to rebuild the project so the editor knows
about the new signal.

[Signal]
public delegate void HitEventHandler();
```

This defines a custom signal called "hit" that we will have our player emit (send out) when it collides with an enemy. We will use `Area2D` to detect the collision. Select the `Player` node and click the "Node" tab next to the Inspector tab to see the list of signals the player can emit:


Inspector
Node
History
Signals
Groups
Filter Signals
player.gd
hit()
Area2D
area_entered(area: Area2D)
area_exited(area: Area2D)
area_shape_entered(area_ri...
area_shape_exited(area_rid:...
body_entered(body: Node2D)
body_exited(body: Node2D)

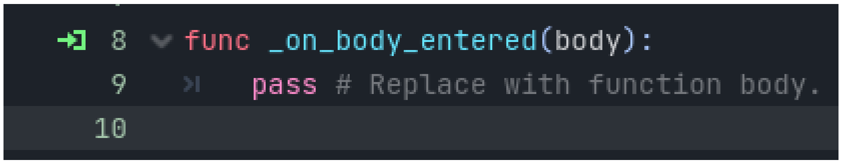
Notice our custom "hit" signal is there as well! Since our enemies are going to be `RigidBody2D` nodes, we want the `body_entered(body: Node2D)` signal. This signal will be emitted when a body contacts the player. Click "Connect.." and the "Connect a Signal" window appears.

Godot will create a function with that exact name directly in script for you. You don't need to change the default settings right now.

Warning

If you're using an external text editor (for example, Visual Studio Code), a bug currently prevents Godot from doing so. You'll be sent to your external editor, but the new function won't be there.

In this case, you'll need to write the function yourself into the Player's script file.



```
func _on_body_entered(body):
	pass # Replace with function body.
```

Note the green icon indicating that a signal is connected to this function; this does not mean the function exists, only that the signal will attempt to connect to a function with that name, so double-check that the spelling of the function matches exactly!

Next, add this code to the function:

GDScript

```
func _on_body_entered(body):
    hide() # Player disappears after being hit.
    hit.emit()
    # Must be deferred as we can't change physics properties on
a physics callback.
    $CollisionShape2D.set_deferred("disabled", true)
```

C#

```
// We also specified this function name in PascalCase in the
editor's connection window.
private void OnBodyEntered(Node2D body)
{
    Hide(); // Player disappears after being hit.
    EmitSignal(SignalName.Hit);
    // Must be deferred as we can't change physics properties
on a physics callback.
    GetNode<CollisionShape2D>
("CollisionShape2D").SetDeferred(CollisionShape2D.PropertyName.
Disabled, true);
}
```

Each time an enemy hits the player, the signal is going to be emitted. We need to disable the player's collision so that we don't trigger the `hit` signal more than once.

Note

Disabling the area's collision shape can cause an error if it happens in the middle of the engine's collision processing. Using `set_deferred()` tells Godot to wait to disable the shape until it's safe to do so.

The last piece is to add a function we can call to reset the player when starting a new game.

GDScript

```
func start(pos):
    position = pos
    show()
    $CollisionShape2D.disabled = false
```

C#

```
public void Start(Vector2 position)
{
    Position = position;
    Show();
    GetNode<CollisionShape2D>("CollisionShape2D").Disabled =
false;
}
```

With the player working, we'll work on the enemy in the next lesson.

Creating the enemy

Now it's time to make the enemies our player will have to dodge. Their behavior will not be very complex: mobs will spawn randomly at the edges of the screen, choose a random direction, and move in a straight line.

We'll create a `Mob` scene, which we can then *instance* to create any number of independent mobs in the game.

Node setup

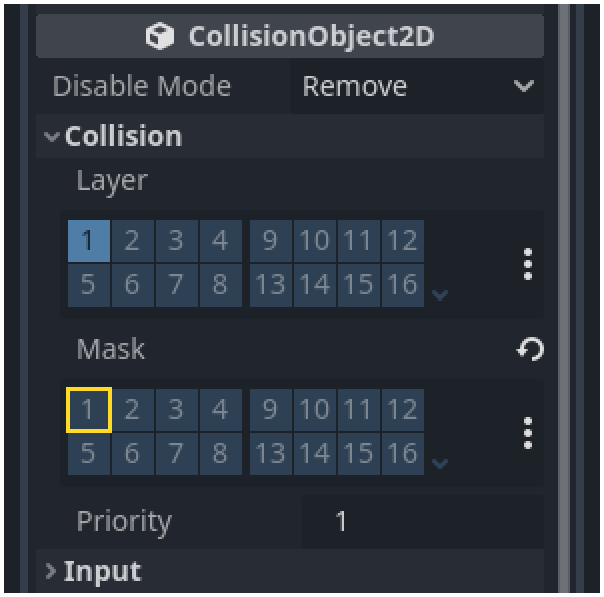
Click Scene -> New Scene from the top menu and add the following nodes:

- RigidBody2D (named `Mob`)
    - AnimatedSprite2D
    - CollisionShape2D
    - VisibleOnScreenNotifier2D

Don't forget to set the children so they can't be selected, like you did with the Player scene.

Select the `Mob` node and set its `Gravity Scale` property in the RigidBody2D section of the inspector to `0`. This will prevent the mob from falling downwards.

In addition, under the CollisionObject2D section just beneath the **RigidBody2D** section, expand the **Collision** group and uncheck the `1` inside the `Mask` property. This will ensure the mobs do not collide with each other.


CollisionObject2D

Disable Mode: Remove

Collision

Layer: 1

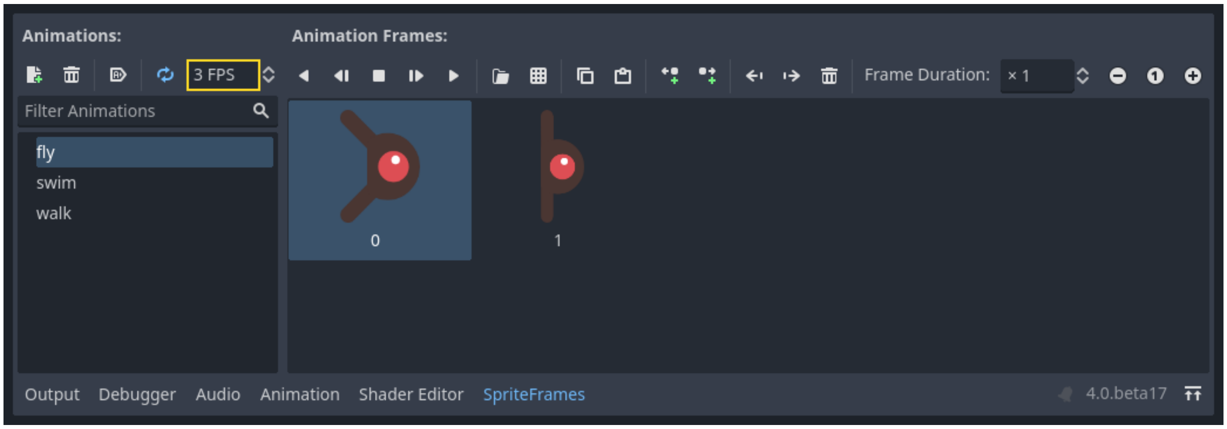
Mask

Priority: 1

1 2 3 4 9 10 11 12
5 6 7 8 13 14 15 16
1 2 3 4 9 10 11 12
5 6 7 8 13 14 15 16
Input

Set up the AnimatedSprite2D like you did for the player. This time, we have 3 animations: `fly`, `swim`, and `walk`. There are two images for each animation in the art folder.

The `Animation Speed` property has to be set for each individual animation. Adjust it to `3` for all 3 animations.


Animations: 3 FPS

Animation Frames:

Frame Duration: × 1

Filter Animations
fly
swim
walk
0
1
Output
Debugger
Audio
Animation
Shader Editor
SpriteFrames
4.0.beta17

You can use the "Play Animation" buttons on the right of the `Animation Speed` input field to preview your animations.

We'll select one of these animations randomly so that the mobs will have some variety.

Like the player images, these mob images need to be scaled down. Set the `AnimatedSprite2D`'s `Scale` property to `(0.75, 0.75)`.

As in the `Player` scene, add a `CapsuleShape2D` for the collision. To align the shape with the image, you'll need to set the `Rotation` property to `90` (under "Transform" in the Inspector).

Save the scene.

Enemy script

Add a script to the `Mob` like this:

GDScript

```
extends RigidBody2D
```

C#

```
using Godot;

public partial class Mob : RigidBody2D
{
    // Don't forget to rebuild the project.
}
```

Now let's look at the rest of the script. In `_ready()` we play the animation and randomly choose one of the three animation types:

GDScript

```
func _ready():
    var mob_types =
$AnimatedSprite2D.sprite_frames.get_animation_names()
    $AnimatedSprite2D.play(mob_types[randi() %
mob_types.size()])
```

C#

```
public override void _Ready()
{
    var animatedSprite2D = GetNode<AnimatedSprite2D>
("AnimatedSprite2D");
    string[] mobTypes =
animatedSprite2D.SpriteFrames.GetAnimationNames();
    animatedSprite2D.Play(mobTypes[GD.Randi() %
mobTypes.Length]);
}
```

First, we get the list of animation names from the AnimatedSprite2D's `sprite_frames` property. This returns an Array containing all three animation names: `["walk", "swim", "fly"]`.

We then need to pick a random number between `0` and `2` to select one of these names from the list (array indices start at `0`). `randi() % n` selects a random integer between `0` and `n-1`.

The last piece is to make the mobs delete themselves when they leave the screen. Connect the `screen_exited()` signal of the `VisibleOnScreenNotifier2D` node to the `Mob` and add this code:

GDScript

```
func _on_visible_on_screen_notifier_2d_screen_exited():
    queue_free()
```

C#

```
// We also specified this function name in PascalCase in the
editor's connection window.
private void OnVisibleOnScreenNotifier2DScreenExited()
{
    QueueFree();
}
```

This completes the *Mob* scene.

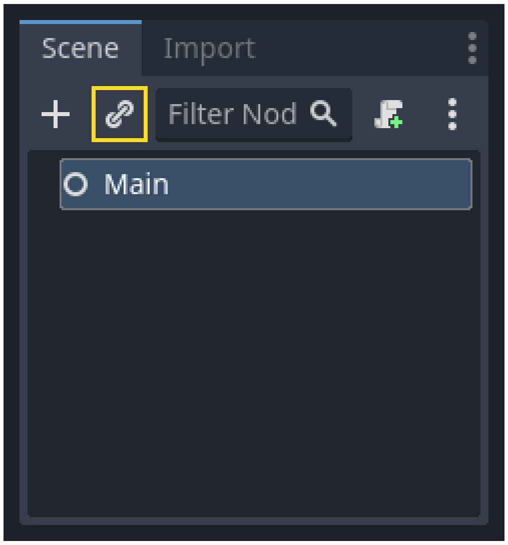
With the player and enemies ready, in the next part, we'll bring them together in a new scene. We'll make enemies spawn randomly around the game board and move forward, turning our project into a playable game.

The main game scene

Now it's time to bring everything we did together into a playable game scene.

Create a new scene and add a Node named `Main`. (The reason we are using Node instead of Node2D is because this node will be a container for handling game logic. It does not require 2D functionality itself.)

Click the **Instance** button (represented by a chain link icon) and select your saved `player.tscn`.



Now, add the following nodes as children of `Main`, and name them as shown:

- Timer (named `MobTimer`) - to control how often mobs spawn
- Timer (named `ScoreTimer`) - to increment the score every second
- Timer (named `StartTimer`) - to give a delay before starting

- Marker2D (named `StartPosition`) - to indicate the player's start position

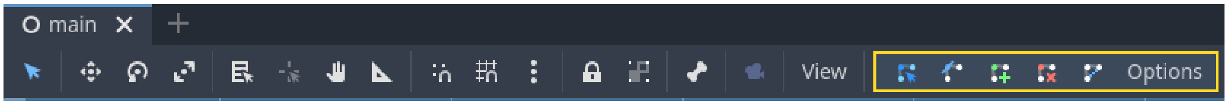
Set the `Wait Time` property of each of the `Timer` nodes as follows (values are in seconds):

- `MobTimer`: `0.5`
- `ScoreTimer`: `1`
- `StartTimer`: `2`

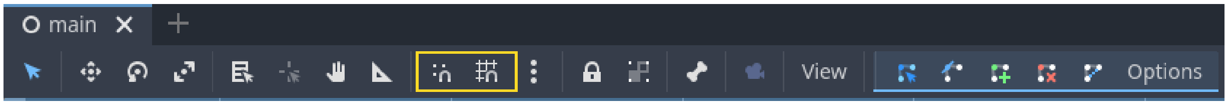
In addition, set the `One Shot` property of `StartTimer` to "On" and set `Position` of the `StartPosition` node to `(240, 450)`.

Spawning mobs

The Main node will be spawning new mobs, and we want them to appear at a random location on the edge of the screen. Add a Path2D node named `MobPath` as a child of `Main`. When you select `Path2D`, you will see some new buttons at the top of the editor:

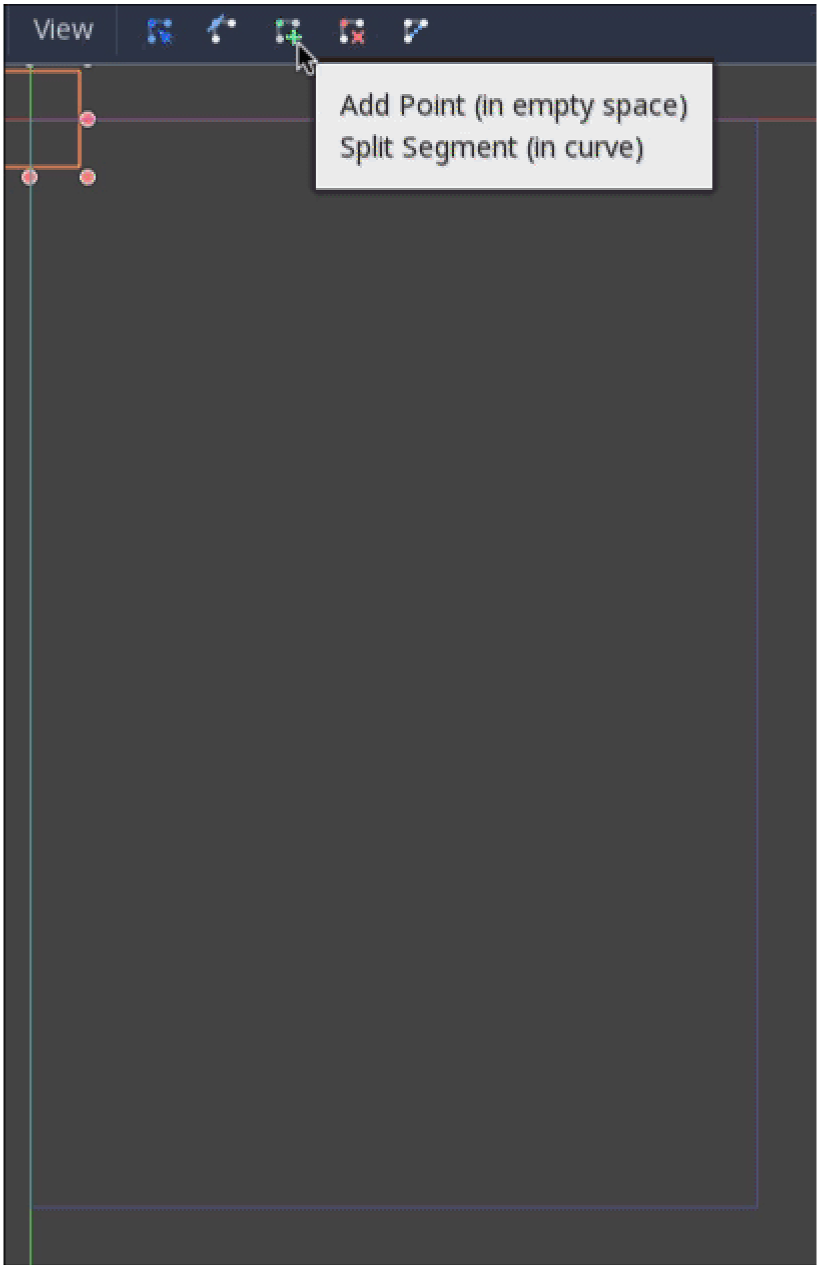


Select the middle one ("Add Point") and draw the path by clicking to add the points at the corners shown. To have the points snap to the grid, make sure "Use Grid Snap" and "Use Smart Snap" are both selected. These options can be found to the left of the "Lock" button, appearing as a magnet next to some dots and intersecting lines, respectively.



Important

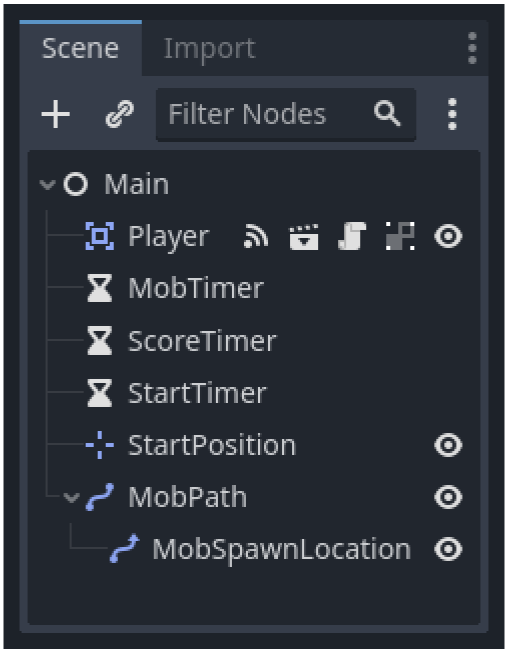
Draw the path in *clockwise* order, or your mobs will spawn pointing *outwards* instead of *inwards*!


View
Add Point (in empty space)
Split Segment (in curve)

After placing point `4` in the image, click the "Close Curve" button and your curve will be complete.

Now that the path is defined, add a PathFollow2D node as a child of `MobPath` and name it `MobSpawnLocation`. This node will automatically rotate and follow the path as it moves, so we can use it to select a random position and direction along the path.

Your scene should look like this:


Scene
Import
Filter Nodes
Main
Player
MobTimer
ScoreTimer
StartTimer
StartPosition
MobPath
MobSpawnLocation

Main script

Add a script to `Main`. At the top of the script, we use `@export var mob_scene: PackedScene` to allow us to choose the Mob scene we want to instance.

GDScript

```
extends Node

@export var mob_scene: PackedScene
var score
```

C#

```
using Godot;

```

```
public partial class Main : Node
{
    // Don't forget to rebuild the project so the editor knows
about the new export variable.

    [Export]
    public PackedScene MobScene { get; set; }

    private int _score;
}
```

Click the `Main` node and you will see the `Mob Scene` property in the Inspector under "Main.gd".

You can assign this property's value in two ways:

- Drag `mob.tscn` from the "FileSystem" dock and drop it in the **Mob Scene** property.
- Click the down arrow next to "[empty]" and choose "Load". Select `mob.tscn`.

Next, select the instance of the `Player` scene under `Main` node in the Scene dock, and access the Node dock on the sidebar. Make sure to have the Signals tab selected in the Node dock.

You should see a list of the signals for the `Player` node. Find and double-click the `hit` signal in the list (or right-click it and select "Connect..."). This will open the signal connection dialog. We want to make a new function named `game_over`, which will handle what needs to happen when a game ends. Type "game_over" in the "Receiver Method" box at the bottom of the signal connection dialog and click "Connect". You are aiming to have the `hit` signal emitted from `Player` and handled in the `Main` script. Add the following code to the new function, as well as a `new_game` function that will set everything up for a new game:

GDScript

```
func game_over():
    $ScoreTimer.stop()
    $MobTimer.stop()

func new_game():
    score = 0
    $Player.start($StartPosition.position)
    $StartTimer.start()
```

C#

```
public void GameOver()
{
    GetNode<Timer>("MobTimer").Stop();
    GetNode<Timer>("ScoreTimer").Stop();
}

public void NewGame()
{
    _score = 0;

    var player = GetNode<Player>("Player");
    var startPosition = GetNode<Marker2D>("StartPosition");
    player.Start(startPosition.Position);

    GetNode<Timer>("StartTimer").Start();
}
```

Now connect the `timeout()` signal of each of the Timer nodes (`StartTimer`, `ScoreTimer`, and `MobTimer`) to the main script. `StartTimer` will start the other two timers. `ScoreTimer` will increment the score by 1.

GDScript

```
func _on_score_timer_timeout():
    score += 1

func _on_start_timer_timeout():
    $MobTimer.start()
    $ScoreTimer.start()
```

C#

```
// We also specified this function name in PascalCase in the
editor's connection window.
private void OnScoreTimerTimeout()
{
    _score++;
}

// We also specified this function name in PascalCase in the
editor's connection window.
private void OnStartTimerTimeout()
{
    GetNode<Timer>("MobTimer").Start();
    GetNode<Timer>("ScoreTimer").Start();
}
```

In `_on_mob_timer_timeout()`, we will create a mob instance, pick a random starting location along the `Path2D`, and set the mob in motion. The `PathFollow2D` node will automatically rotate as it follows the path, so we will use that to select the mob's direction as well as its position. When we spawn a mob, we'll pick a random value between `150.0` and `250.0` for how fast each mob will move (it would be boring if they were all moving at the same speed).

Note that a new instance must be added to the scene using `add_child()`.

GDScript

```
func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on Path2D.
    var mob_spawn_location = $MobPath/MobSpawnLocation
    mob_spawn_location.progress_ratio = randf()

    # Set the mob's direction perpendicular to the path
direction.
```

```
    var direction = mob_spawn_location.rotation + PI / 2

    # Set the mob's position to a random location.
    mob.position = mob_spawn_location.position

    # Add some randomness to the direction.
    direction += randf_range(-PI / 4, PI / 4)
    mob.rotation = direction

    # Choose the velocity for the mob.
    var velocity = Vector2(randf_range(150.0, 250.0), 0.0)
    mob.linear_velocity = velocity.rotated(direction)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)
```

C#

```
// We also specified this function name in PascalCase in the
editor's connection window.
private void OnMobTimerTimeout()
{
    // Note: Normally it is best to use explicit types rather
than the `var`
    // keyword. However, var is acceptable to use here because
the types are
    // obviously Mob and PathFollow2D, since they appear later
on the line.

    // Create a new instance of the Mob scene.
    Mob mob = MobScene.Instantiate<Mob>();

    // Choose a random location on Path2D.
    var mobSpawnLocation = GetNode<PathFollow2D>
("MobPath/MobSpawnLocation");
    mobSpawnLocation.ProgressRatio = GD.Randf();

    // Set the mob's direction perpendicular to the path
direction.
    float direction = mobSpawnLocation.Rotation + Mathf.Pi / 2;

    // Set the mob's position to a random location.
    mob.Position = mobSpawnLocation.Position;

    // Add some randomness to the direction.
    direction += (float)GD.RandRange(-Mathf.Pi / 4, Mathf.Pi /
4);
    mob.Rotation = direction;

    // Choose the velocity.
    var velocity = new Vector2((float)GD.RandRange(150.0,
250.0), 0);
    mob.LinearVelocity = velocity.Rotated(direction);

    // Spawn the mob by adding it to the Main scene.
    AddChild(mob);
}
```

Important

Why `PI`? In functions requiring angles, Godot uses *radians*, not degrees. Pi represents a half turn in radians, about `3.1415` (there is also `TAU` which is equal to `2 * PI`). If you're more comfortable working with degrees, you'll need to use the `deg_to_rad()` and `rad_to_deg()` functions to convert between the two.

Testing the scene

Let's test the scene to make sure everything is working. Add this `new_game` call to `_ready()`:

GDScript

```
func _ready():
    new_game()
```

C#

```
public override void _Ready()
{
    NewGame();
}
```

Let's also assign `Main` as our "Main Scene" - the one that runs automatically when the game launches. Press the "Play" button and select `main.tscn` when prompted.

Tip

If you had already set another scene as the "Main Scene", you can right click `main.tscn` in the FileSystem dock and select "Set As Main Scene".

You should be able to move the player around, see mobs spawning, and see the player disappear when hit by a mob.

When you're sure everything is working, remove the call to `new_game()` from `_ready()` and replace it with `pass`.

What's our game lacking? Some user interface. In the next lesson, we'll add a title screen and display the player's score.

Heads up display

The final piece our game needs is a User Interface (UI) to display things like score, a "game over" message, and a restart button.

Create a new scene, click the "Other Node" button and add a CanvasLayer node named `HUD`. "HUD" stands for "heads-up display", an informational display that appears as an overlay on top of the game view.

The CanvasLayer node lets us draw our UI elements on a layer above the rest of the game, so that the information it displays isn't covered up by any game elements like the player or mobs.

The HUD needs to display the following information:

- Score, changed by `ScoreTimer`.
- A message, such as "Game Over" or "Get Ready!"
- A "Start" button to begin the game.

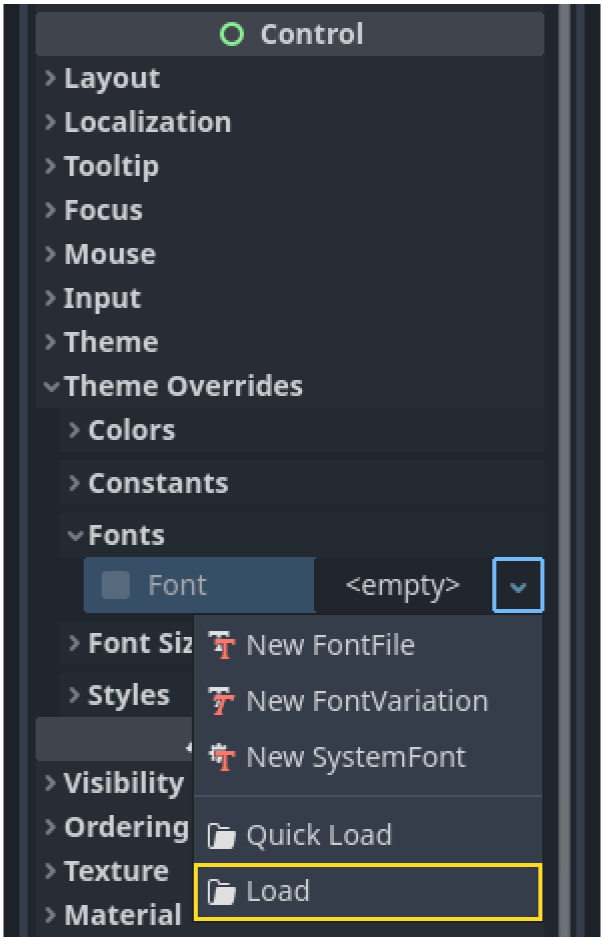
The basic node for UI elements is Control. To create our UI, we'll use two types of Control nodes: Label and Button.

Create the following as children of the `HUD` node:

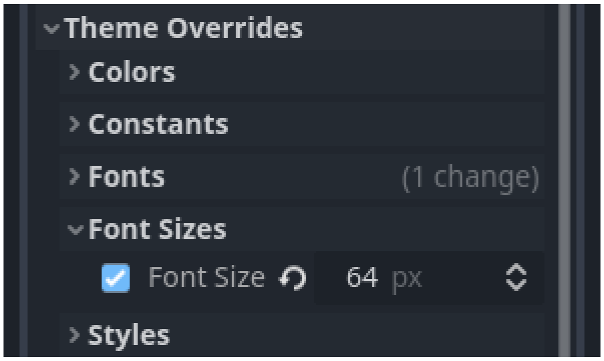
- Label named `ScoreLabel`.
- Label named `Message`.
- Button named `StartButton`.
- Timer named `MessageTimer`.

Click on the `ScoreLabel` and type a number into the `Text` field in the Inspector. The default font for `Control` nodes is small and doesn't scale well. There is a font file included in the game assets called "Xolonium-Regular.ttf". To use this font, do the following:

Under "Theme Overrides > Fonts", choose "Load" and select the "Xolonium-Regular.ttf" file.


Control
Layout
Localization
Tooltip
Focus
Mouse
Input
Theme
Theme Overrides
Colors
Constants
Fonts
Font
<empty>
Font Siz
Styles
Visibility
Ordering
Texture
Material
New FontFile
New FontVariation
New SystemFont
Quick Load
Load

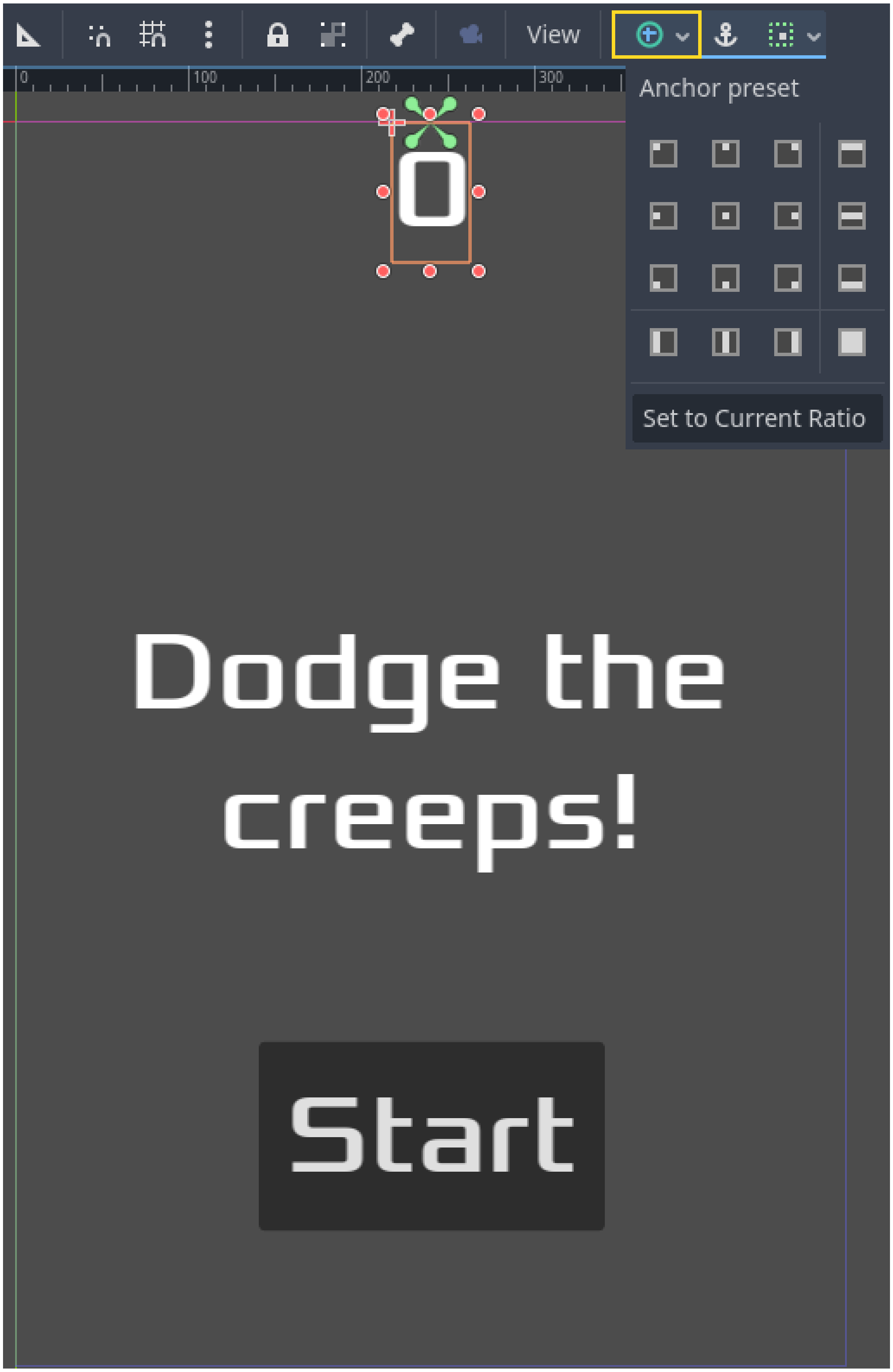
The font size is still too small, increase it to `64` under "Theme Overrides > Font Sizes". Once you've done this with the `ScoreLabel`, repeat the changes for the `Message` and `StartButton` nodes.


Theme Overrides
Colors
Constants
Fonts
(1 change)
Font Sizes
Font Size
64 px
Styles

Note

Anchors: `Control` nodes have a position and size, but they also have anchors. Anchors define the origin - the reference point for the edges of the node.

Arrange the nodes as shown below. You can drag the nodes to place them manually, or for more precise placement, use "Anchor Presets".


View
0
100
200
300
Anchor preset
Set to Current Ratio
0
Dodge the creeps!
Start

ScoreLabel

1. Add the text `0`.
2. Set the "Horizontal Alignment" and "Vertical Alignment" to `Center`.
3. Choose the "Anchor Preset" `Center Top`.

Message

1. Add the text `Dodge the Creeps!`.
2. Set the "Horizontal Alignment" and "Vertical Alignment" to `Center`.
3. Set the "Autowrap Mode" to `Word`, otherwise the label will stay on one line.
4. Under "Control - Layout/Transform" set "Size X" to `480` to use the entire width of the screen.
5. Choose the "Anchor Preset" `Center`.

StartButton

1. Add the text `Start`.
2. Under "Control - Layout/Transform", set "Size X" to `200` and "Size Y" to `100` to add a little bit more padding between the border and text.
3. Choose the "Anchor Preset" `Center Bottom`.
4. Under "Control - Layout/Transform", set "Position Y" to `580`.

On the `MessageTimer`, set the `Wait Time` to `2` and set the `One Shot` property to "On".

Now add this script to `HUD`:

GDScript

```
extends CanvasLayer
```

```
# Notifies `Main` node that the button has been pressed
signal start_game
```

C#

```
using Godot;

public partial class HUD : CanvasLayer
{
    // Don't forget to rebuild the project so the editor knows
about the new signal.

    [Signal]
    public delegate void StartGameEventHandler();
}
```

We now want to display a message temporarily, such as "Get Ready", so we add the following code

GDScript

```
func show_message(text):
    $Message.text = text
    $Message.show()
    $MessageTimer.start()
```

C#

```
public void ShowMessage(string text)
{
    var message = GetNode<Label>("Message");
    message.Text = text;
    message.Show();

    GetNode<Timer>("MessageTimer").Start();
}
```

We also need to process what happens when the player loses. The code below will show "Game Over" for 2 seconds, then return to the title screen and, after a brief pause, show the "Start" button.

GDScript

```
func show_game_over():
    show_message("Game Over")
    # Wait until the MessageTimer has counted down.
    await $MessageTimer.timeout

    $Message.text = "Dodge the Creeps!"
    $Message.show()
    # Make a one-shot timer and wait for it to finish.
    await get_tree().create_timer(1.0).timeout
    $StartButton.show()
```

C#

```
async public void ShowGameOver()
{
    ShowMessage("Game Over");

    var messageTimer = GetNode<Timer>("MessageTimer");
    await ToSignal(messageTimer, Timer.SignalName.Timeout);

    var message = GetNode<Label>("Message");
    message.Text = "Dodge the Creeps!";
    message.Show();

    await ToSignal(GetTree().CreateTimer(1.0),
SceneTreeTimer.SignalName.Timeout);
    GetNode<Button>("StartButton").Show();
}
```

This function is called when the player loses. It will show "Game Over" for 2 seconds, then return to the title screen and, after a brief pause, show the "Start" button.

Note

When you need to pause for a brief time, an alternative to using a Timer node is to use the SceneTree's `create_timer()` function. This can be very useful to add delays such as in the above code, where we want to wait some time before showing the "Start" button.

Add the code below to `HUD` to update the score

GDScript

```
func update_score(score):
    $ScoreLabel.text = str(score)
```

C#

```
public void UpdateScore(int score)
{
    GetNode<Label>("ScoreLabel").Text = score.ToString();
}
```

Connect the `pressed()` signal of `StartButton` and the `timeout()` signal of `MessageTimer` to the `HUD` node, and add the following code to the new functions:

GDScript

```
func _on_start_button_pressed():
    $StartButton.hide()
    start_game.emit()

func _on_message_timer_timeout():
    $Message.hide()
```

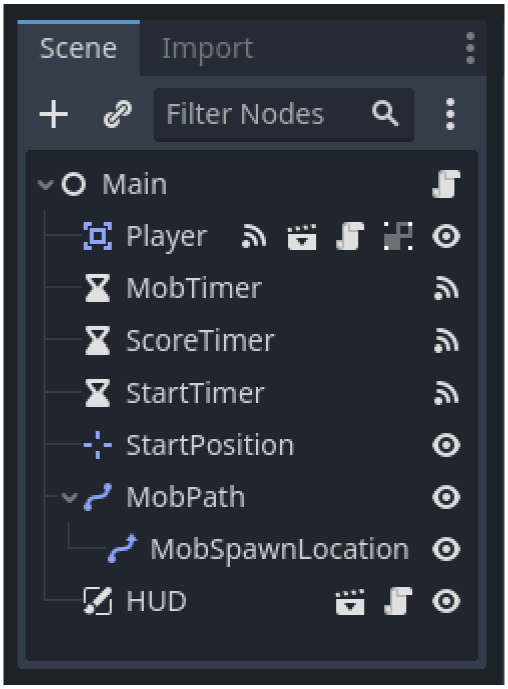
C#

```
// We also specified this function name in PascalCase in the
editor's connection window.
private void OnStartButtonPressed()
{
    GetNode<Button>("StartButton").Hide();
    EmitSignal(SignalName.StartGame);
}
```

```
// We also specified this function name in PascalCase in the
editor's connection window.
private void OnMessageTimerTimeout()
{
    GetNode<Label>("Message").Hide();
}
```

Connecting HUD to Main

Now that we're done creating the `HUD` scene, go back to `Main`. Instance the `HUD` scene in `Main` like you did the `Player` scene. The scene tree should look like this, so make sure you didn't miss anything:


Scene
Import
Filter Nodes
Main
Player
MobTimer
ScoreTimer
StartTimer
StartPosition
MobPath
MobSpawnLocation
HUD

Now we need to connect the `HUD` functionality to our `Main` script. This requires a few additions to the `Main` scene:

In the Node tab, connect the HUD's `start_game` signal to the `new_game()` function of the Main node by clicking the "Pick" button in the "Connect a Signal" window and selecting the `new_game()` method or type "new_game" below "Receiver Method" in the window. Verify that the green connection icon now appears next to `func new_game()` in the script.

In `new_game()`, update the score display and show the "Get Ready" message:

GDScript

```
$HUD.update_score(score)
$HUD.show_message("Get Ready")
```

C#

```
var hud = GetNode<HUD>("HUD");
hud.UpdateScore(_score);
hud.ShowMessage("Get Ready!");
```

In `game_over()` we need to call the corresponding `HUD` function:

GDScript

```
$HUD.show_game_over()
```

C#

```
GetNode<HUD>("HUD").ShowGameOver();
```

Finally, add this to `_on_score_timer_timeout()` to keep the display in sync with the changing score:

GDScript

```
$HUD.update_score(score)
```

C#

```
GetNode<HUD>("HUD").UpdateScore(_score);
```

Warning

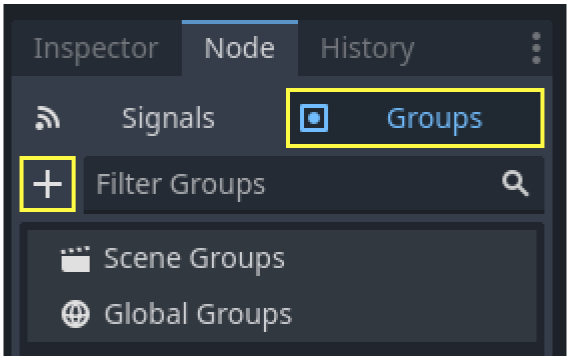
Remember to remove the call to `new_game()` from `_ready()` if you haven't already, otherwise your game will start automatically.

Now you're ready to play! Click the "Play the Project" button.

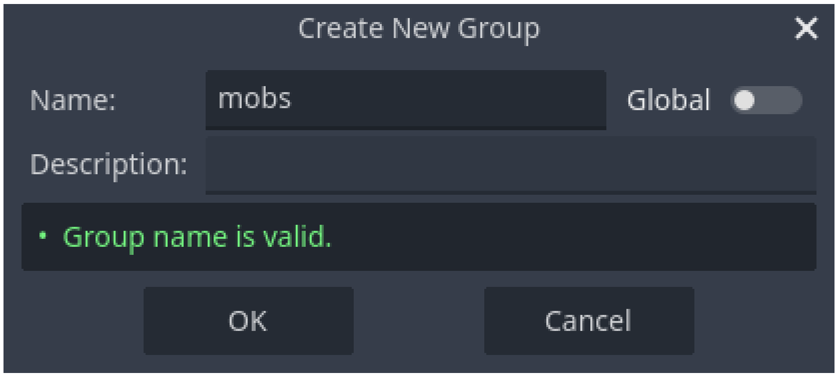
Removing old creeps

If you play until "Game Over" and then start a new game right away, the creeps from the previous game may still be on the screen. It would be better if they all disappeared at the start of a new game. We just need a way to tell *all* the mobs to remove themselves. We can do this with the "group" feature.

In the `Mob` scene, select the root node and click the "Node" tab next to the Inspector (the same place where you find the node's signals). Next to "Signals", click "Groups" to open the group overview and the "+" button to open the "Create New Group" dialog.



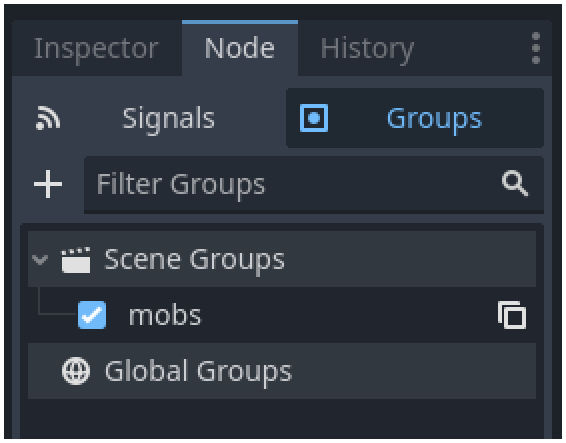
Name the group `mobs` and click "ok" to add a new scene group.


Create New Group
Name:
Global
Description:
OK
Cancel

Name: mobs

• Group name is valid.

Now all mobs will be in the "mobs" group.


Inspector
Node
History
Signals
Groups
Filter Groups
Scene Groups
mobs
Global Groups

We can then add the following line to the `new_game()` function in `Main`:

GDScript

```
get_tree().call_group("mobs", "queue_free")
```

C#

```
// Note that for calling Godot-provided methods with strings,
// we have to use the original Godot snake_case name.
GetTree().CallGroup("mobs", Node.MethodName.QueueFree);
```

The `call_group()` function calls the named function on every node in a group - in this case we are telling every mob to delete itself.

The game's mostly done at this point. In the next and last part, we'll polish it a bit by adding a background, looping music, and some keyboard shortcuts.

Finishing up

We have now completed all the functionality for our game. Below are some remaining steps to add a bit more "juice" to improve the game experience.

Feel free to expand the gameplay with your own ideas.

Background

The default gray background is not very appealing, so let's change its color. One way to do this is to use a ColorRect node. Make it the first node under `Main` so that it will be drawn behind the other nodes. `ColorRect` only has one property: `Color`. Choose a color you like and select "Layout" -> "Anchors Preset" -> "Full Rect" either in the toolbar at the top of the viewport or in the inspector so that it covers the screen.

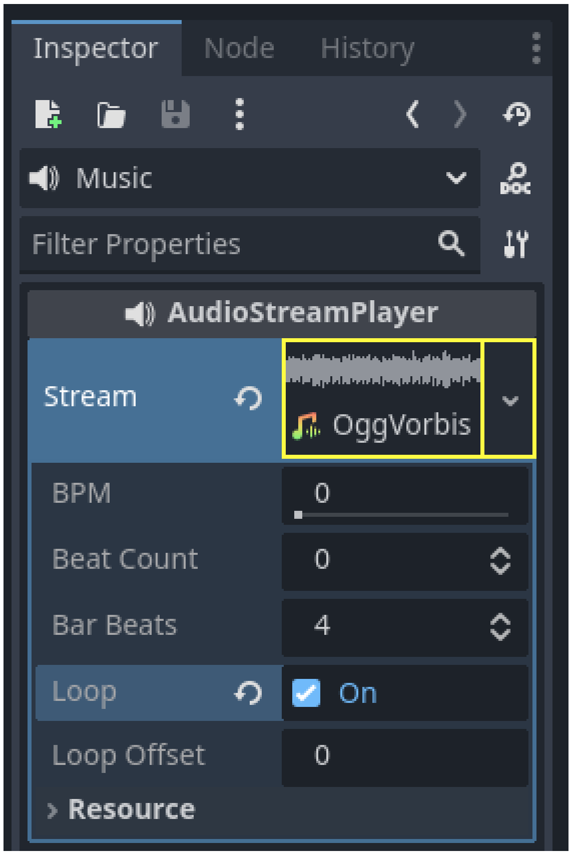
You could also add a background image, if you have one, by using a TextureRect node instead.

Sound effects

Sound and music can be the single most effective way to add appeal to the game experience. In your game's **art** folder, you have two sound files: "House In a Forest Loop.ogg" for background music, and "gameover.wav" for when the player loses.

Add two AudioStreamPlayer nodes as children of `Main`. Name one of them `Music` and the other `DeathSound`. On each one, click on the `Stream` property, select "Load", and choose the corresponding audio file.

All audio is automatically imported with the `Loop` setting disabled. If you want the music to loop seamlessly, click on the Stream file arrow, select `Make Unique`, then click on the Stream file and check the `Loop` box.


Inspector
Node
History
Music
Filter Properties
AudioStreamPlayer
Resource

Stream: OggVorbis
BPM: 0
Beat Count: 0
Bar Beats: 4
Loop: ☑ On
Loop Offset: 0

To play the music, add `$Music.play()` in the `new_game()` function and `$Music.stop()` in the `game_over()` function.

Finally, add `$DeathSound.play()` in the `game_over()` function.

GDScript

```
func game_over():
    ...
    $Music.stop()
    $DeathSound.play()

func new_game():
    ...
    $Music.play()
```

C#

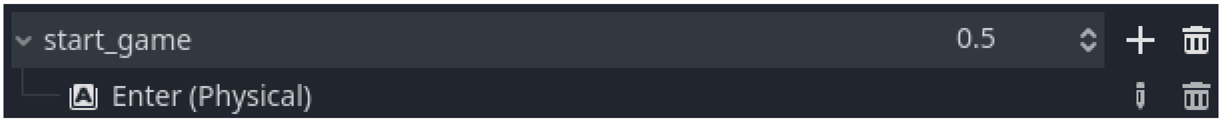
```
public void GameOver()
{
    ...
    GetNode<AudioStreamPlayer>("Music").Stop();
    GetNode<AudioStreamPlayer>("DeathSound").Play();
}

public void NewGame()
{
    ...
    GetNode<AudioStreamPlayer>("Music").Play();
}
```

Keyboard shortcut

Since the game is played with keyboard controls, it would be convenient if we could also start the game by pressing a key on the keyboard. We can do this with the "Shortcut" property of the `Button` node.

In a previous lesson, we created four input actions to move the character. We will create a similar input action to map to the start button.

Select "Project" -> "Project Settings" and then click on the "Input Map" tab. In the same way you created the movement input actions, create a new input action called `start_game` and add a key mapping for the `Enter` key.

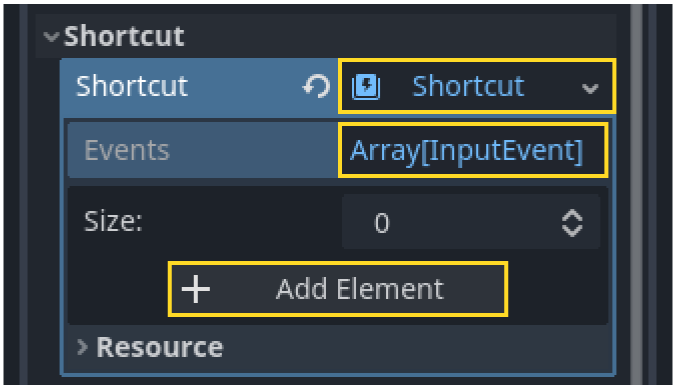


start_game 0.5

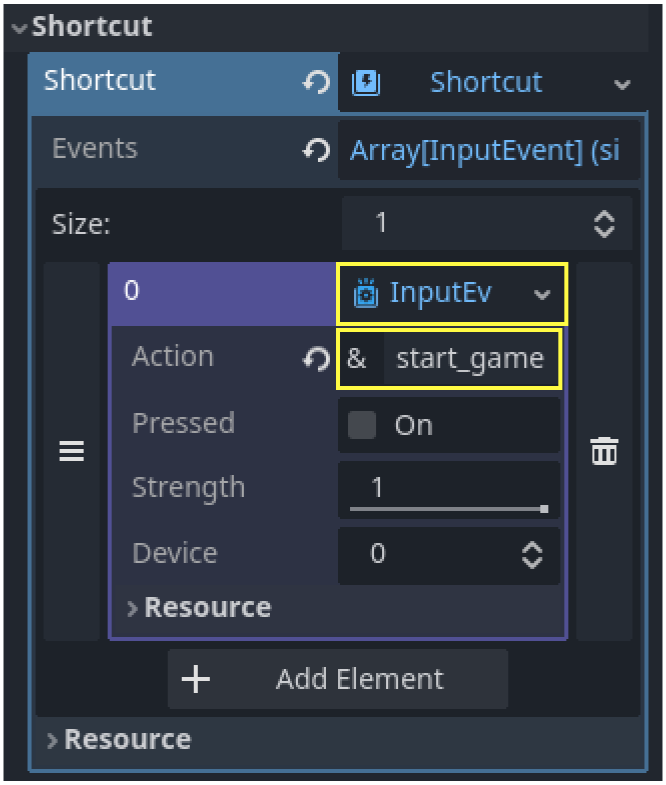
Enter (Physical)

Now would be a good time to add controller support if you have one available. Attach or pair your controller and then under each input action that you wish to add controller support for, click on the "+" button and press the corresponding button, d-pad, or stick direction that you want to map to the respective input action.

In the `HUD` scene, select the `StartButton` and find its **Shortcut** property in the Inspector. Create a new Shortcut resource by clicking within the box, open the **Events** array and add a new array element to it by clicking on **Array[InputEvent] (size 0)**.


Shortcut
Shortcut
Shortcut
Events
Array[InputEvent]
Size:
0
Add Element
Resource

Create a new InputEventAction and name it `start_game`.


Shortcut
Shortcut
Shortcut
0
InputEv
Resource
Add Element
Resource

Events: Array[InputEvent] (si

Size: 1

Action: & start_game

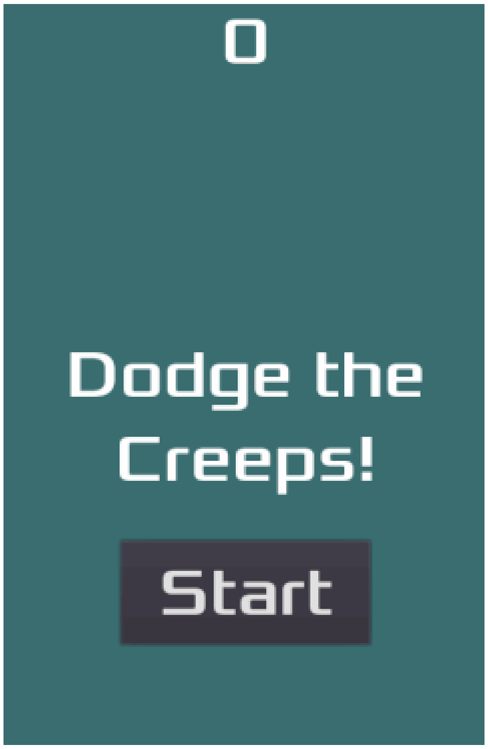
Pressed: On

Strength: 1

Device: 0

Now when the start button appears, you can either click it or press `Enter` to start the game.

And with that, you completed your first 2D game in Godot.



You got to make a player-controlled character, enemies that spawn randomly around the game board, count the score, implement a game over and replay, user interface, sounds, and more. Congratulations!

There's still much to learn, but you can take a moment to appreciate what you achieved.

And when you're ready, you can move on to Your first 3D game to learn to create a complete 3D game from scratch, in Godot.

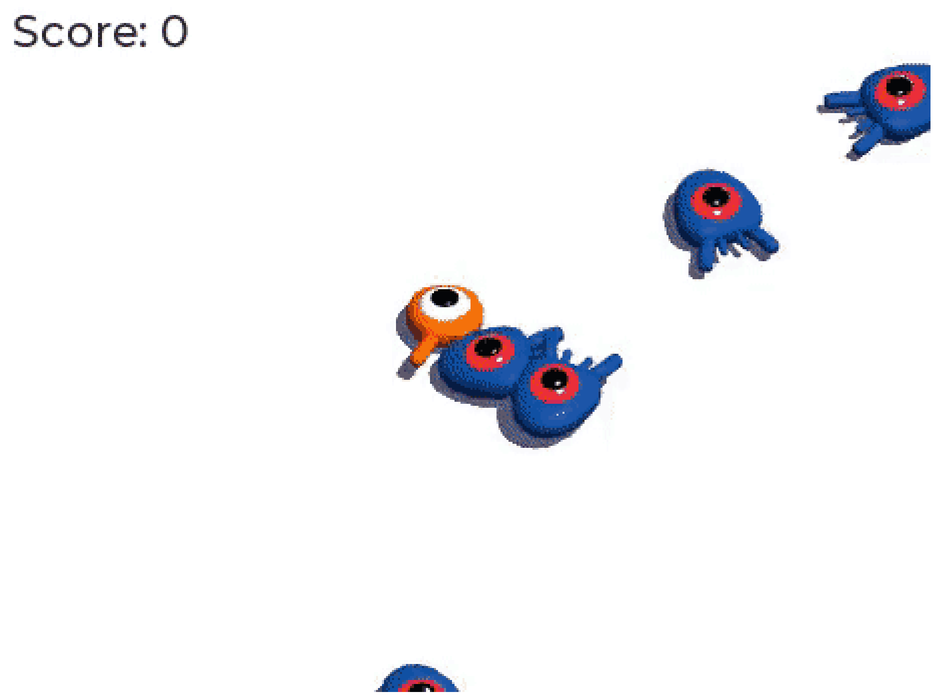
Sharing the finished game with others

If you want people to try out your game without having to install Godot, you'll need to export the project for each operating system you want the game to be playable on. See Exporting projects for instructions.

After exporting the project, compress the exported executable and PCK file (not the raw project files) to a ZIP file, then upload this ZIP file to a file sharing website.

Your first 3D game

In this step-by-step tutorial series, you will create your first complete 3D game with Godot. By the end of the series, you will have a simple yet finished project of your own like the animated gif below.



The game we'll code here is similar to [Your first 2D game](), with a twist: you can now jump and your goal is to squash the creeps. This way, you will both **recognize patterns** you learned in the previous tutorial and **build upon them** with new code and features.

You will learn to:

- Work with 3D coordinates with a jumping mechanic.
- Use kinematic bodies to move 3D characters and detect when and how they collide.
- Use physics layers and a group to detect interactions with specific entities.

- Code basic procedural gameplay by instancing monsters at regular time intervals.
- Design a movement animation and change its speed at run-time.
- Draw a user interface on a 3D game.

And more.

This tutorial is for beginners who followed the complete getting started series. We'll start slow with detailed instructions and shorten them as we do similar steps. If you're an experienced programmer, you can browse the complete demo's source code here: Squash the Creep source code [https://github.com/godotengine/godot-3d-dodge-the-creeps].

Note

You can follow this series without having done the 2D one. However, if you're new to game development, we recommend you to start with 2D. 3D game code is always more complex and the 2D series will give you foundations to follow along more comfortably.

We prepared some game assets so we can jump straight to the code. You can download them here: Squash the Creeps assets [https://github.com/godotengine/godot-3d-dodge-the-creeps/releases/tag/1.1.0].

We will first work on a basic prototype for the player's movement. We will then add the monsters that we'll spawn randomly around the screen. After that, we'll implement the jump and squashing mechanic before refining the game with some nice animation. We'll wrap up with the score and the retry screen.

Contents

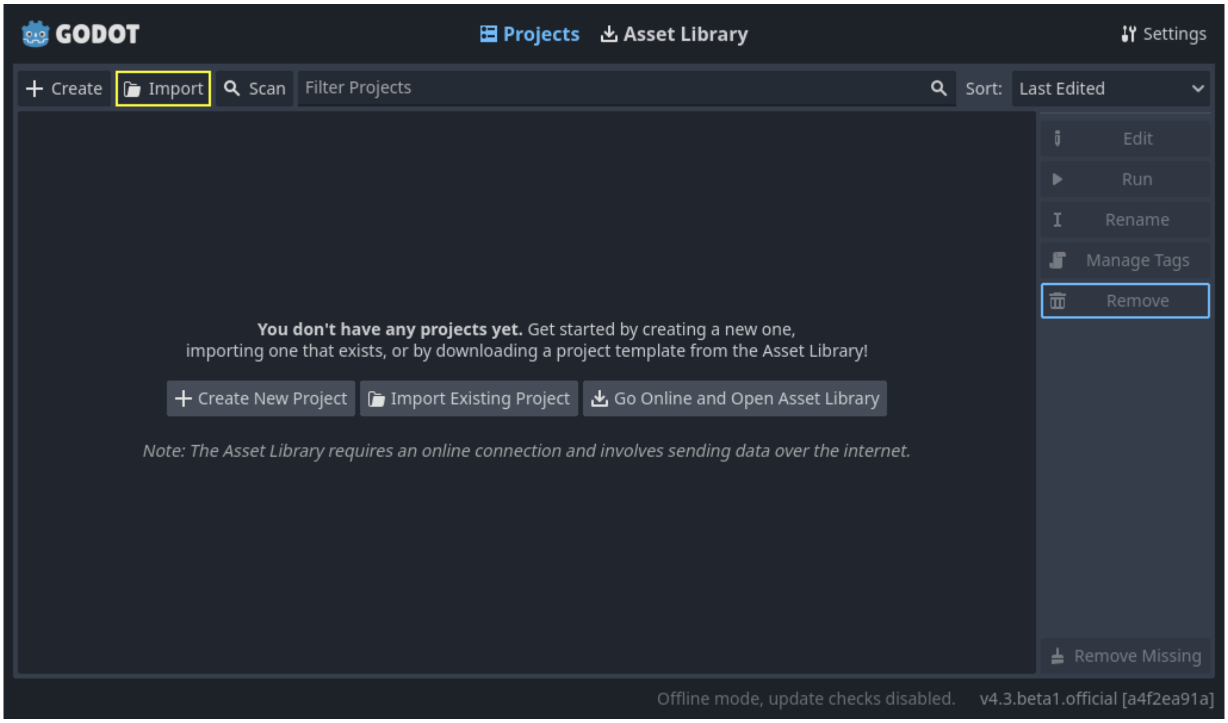
- Setting up the game area
- Player scene and input actions
- Moving the player with code
- Designing the mob scene
- Spawning monsters
- Jumping and squashing monsters
- Killing the player
- Score and replay
- Character animation
- Going further

Setting up the game area

In this first part, we're going to set up the game area. Let's get started by importing the start assets and setting up the game scene.

We've prepared a Godot project with the 3D models and sounds we'll use for this tutorial, linked in the index page. If you haven't done so yet, you can download the archive here: Squash the Creeps assets [https://github.com/godotengine/godot-3d-dodge-the-creeps/releases/tag/1.1.0].

Once you downloaded it, extract the .zip archive on your computer. Open the Godot Project Manager and click the *Import* button.



You don't have any projects yet. Get started by creating a new one, importing one that exists, or by downloading a project template from the Asset Library!

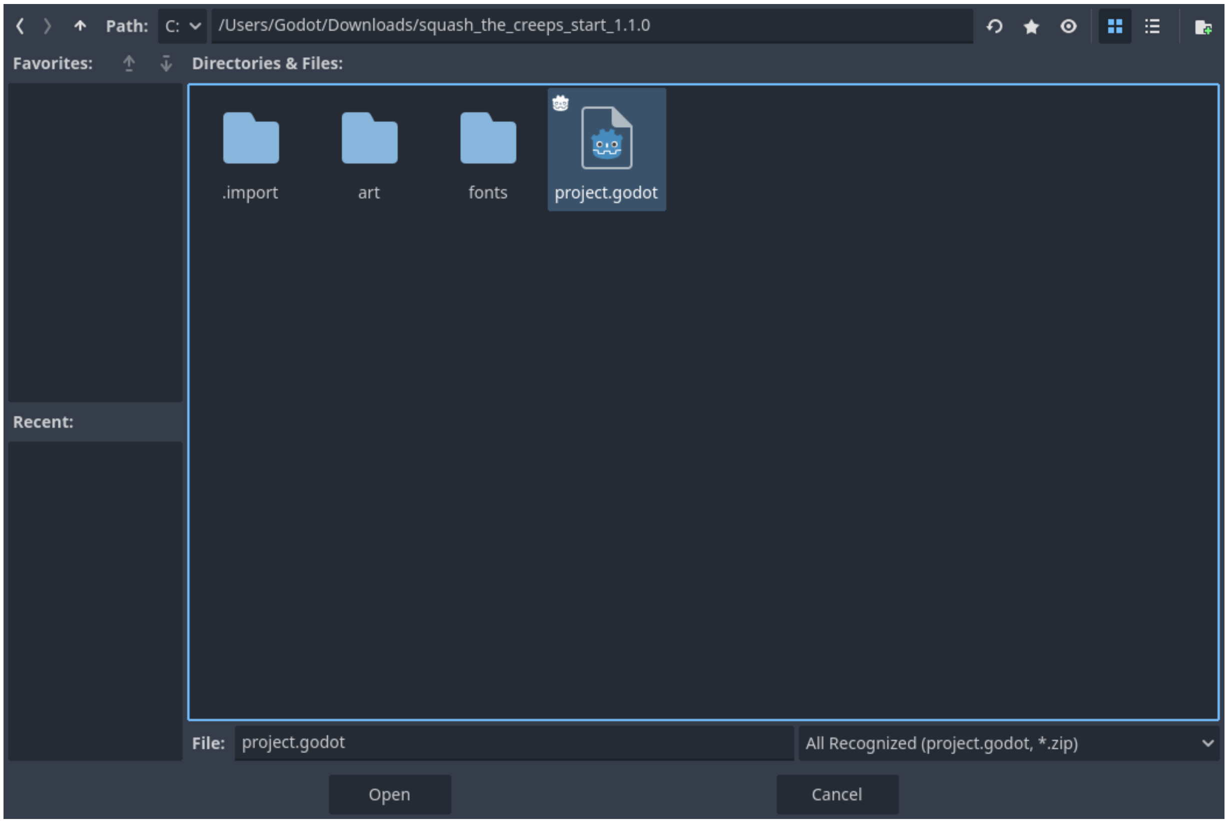
Note: The Asset Library requires an online connection and involves sending data over the internet.

Offline mode, update checks disabled.

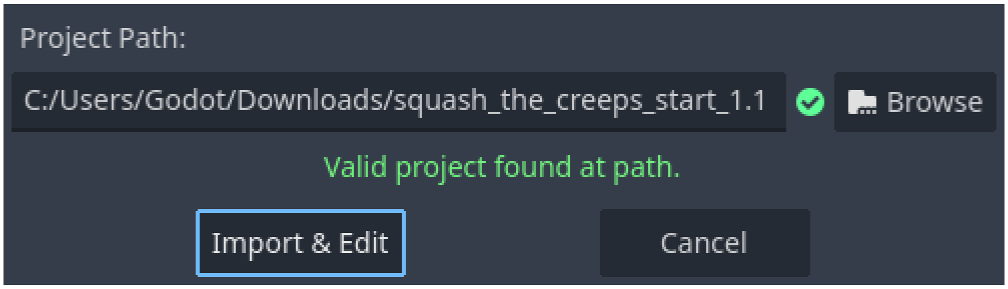
v4.3.beta1.official [a4f2ea91a]

GODOT
Projects
Asset Library
Settings
Create
Import
Scan
Filter Projects
Sort:
Last Edited
Edit
Run
Rename
Manage Tags
Remove
Create New Project
Import Existing Project
Go Online and Open Asset Library
Remove Missing

In the import popup, enter the full path to the freshly created directory `squash_the_creeps_start/`. You can click the *Browse* button on the right to open a file browser and navigate to the `project.godot` file the folder contains.

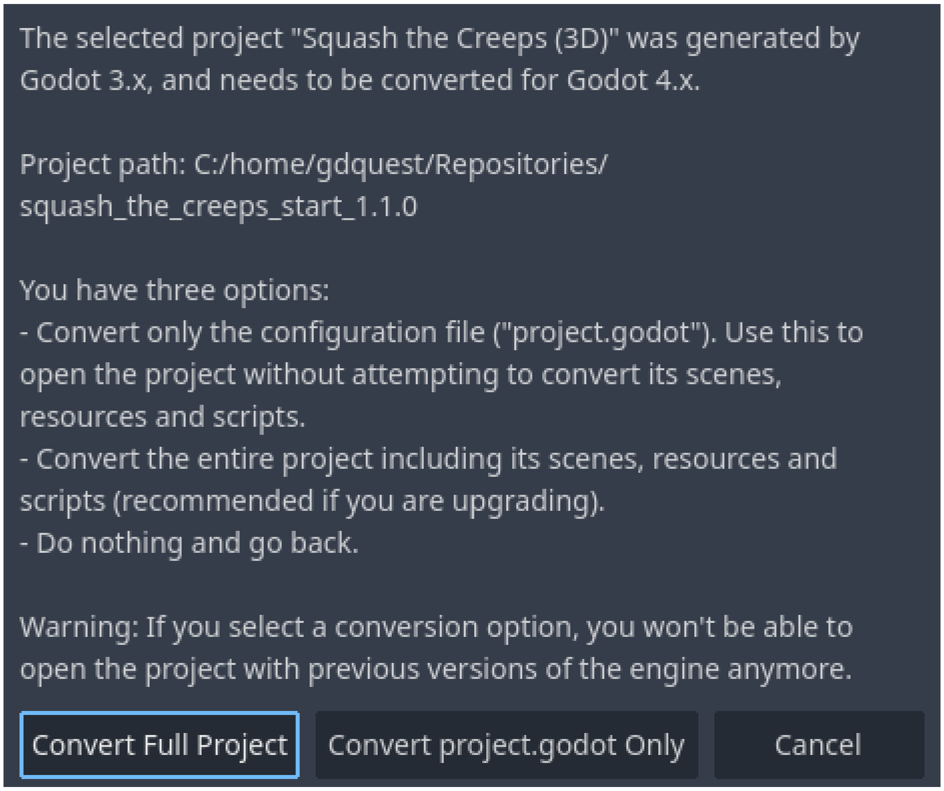

Path: C: /Users/Godot/Downloads/squash_the_creeps_start_1.1.0
Favorites:
Directories & Files:
.import
art
fonts
project.godot
Recent:
File: project.godot
All Recognized (project.godot, *.zip)
Open
Cancel

Click *Import & Edit* to open the project in the editor.


Project Path:
C:/Users/Godot/Downloads/squash_the_creeps_start_1.1
Browse
Import & Edit
Cancel

Valid project found at path.

A window notifying you that the project was generated by an older Godot version may appear. Click *Convert Full Project* to convert the project to your current Godot version.



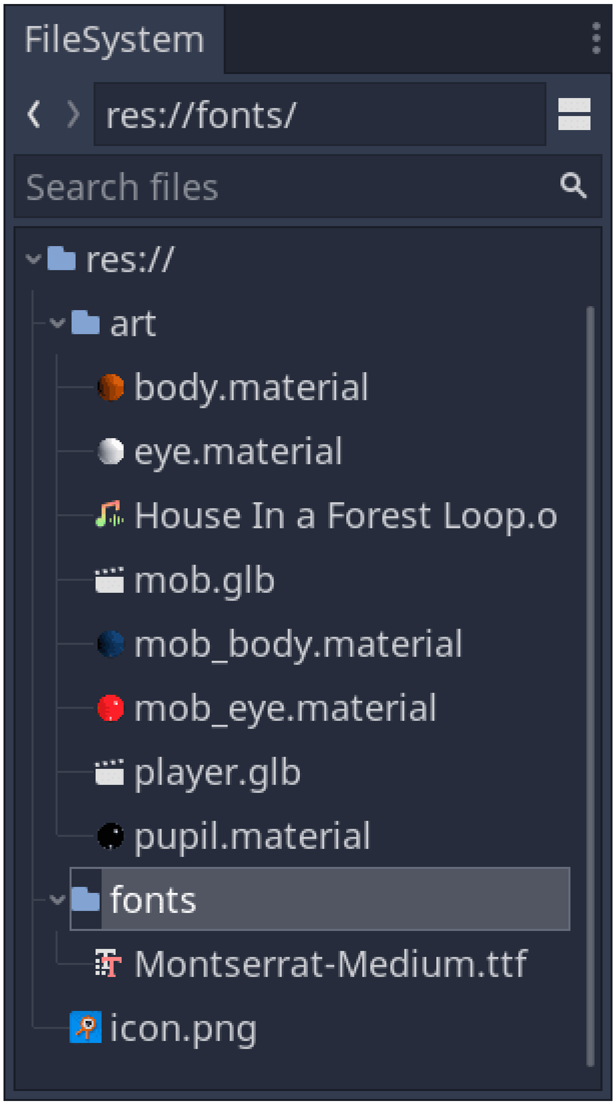
The selected project "Squash the Creeps (3D)" was generated by Godot 3.x, and needs to be converted for Godot 4.x.

Project path: C:/home/gdquest/Repositories/squash_the_creeps_start_1.1.0

You have three options:
- Convert only the configuration file ("project.godot"). Use this to open the project without attempting to convert its scenes, resources and scripts.
- Convert the entire project including its scenes, resources and scripts (recommended if you are upgrading).
- Do nothing and go back.

Warning: If you select a conversion option, you won't be able to open the project with previous versions of the engine anymore.

The start project contains an icon and two folders: `art/` and `fonts/`. There, you will find the art assets and music we'll use in the game.

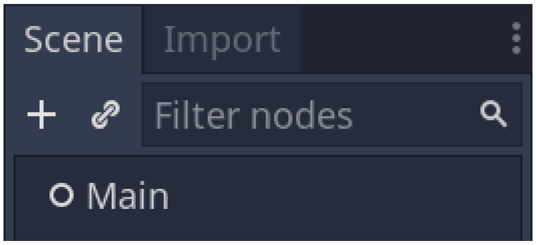


FileSystem
res://fonts/
Search files
res://
art
body.material
eye.material
House In a Forest Loop.o
mob.glb
mob_body.material
mob_eye.material
player.glb
pupil.material
fonts
Montserrat-Medium.ttf
icon.png

There are two 3D models, `player.glb` and `mob.glb`, some materials that belong to these models, and a music track.

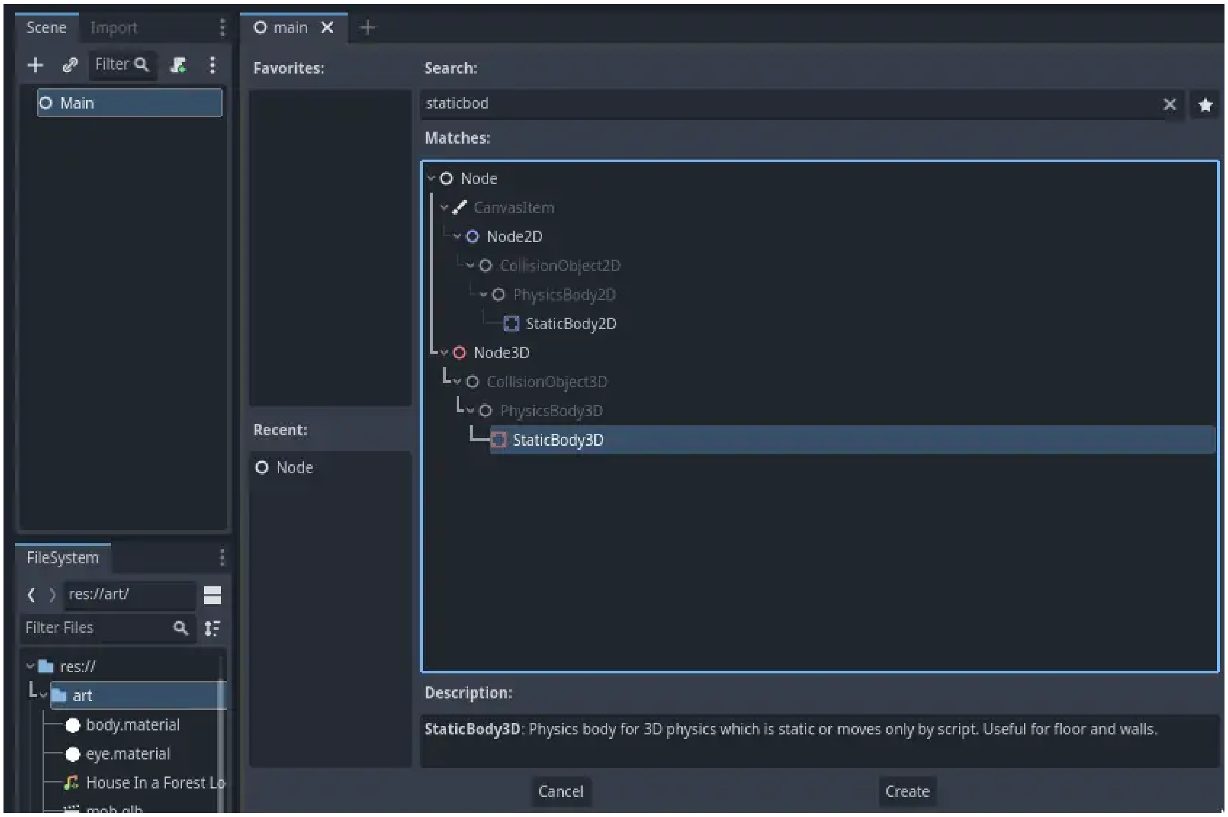
Setting up the playable area

We're going to create our main scene with a plain Node as its root. In the *Scene* dock, click the *Add Child Node* button represented by a "+" icon in the top-left and double-click on *Node*. Name the node `Main`. An alternate method to rename the node is to right-click on *Node* and choose *Rename* (or `F2`). Alternatively, to add a node to the scene, you can press `Ctrl + A` (`Cmd + A` on macOS).


Scene
Import
Filter nodes
Main

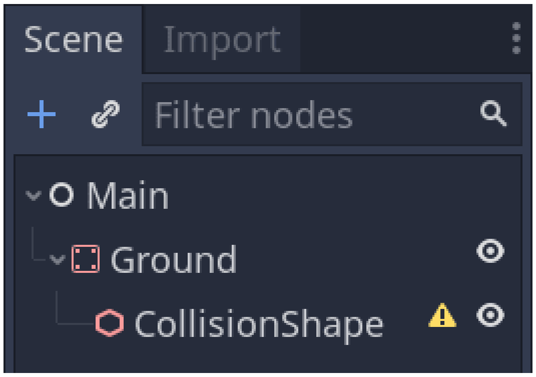
Save the scene as `main.tscn` by pressing `Ctrl + S` (`Cmd + S` on macOS).

We'll start by adding a floor that'll prevent the characters from falling. To create static colliders like the floor, walls, or ceilings, you can use [StaticBody3D]() nodes. They require [CollisionShape3D]() child nodes to define the collision area. With the `Main` node selected, add a [StaticBody3D]() node, then a [CollisionShape3D](). Rename the [StaticBody3D]() to `Ground`.

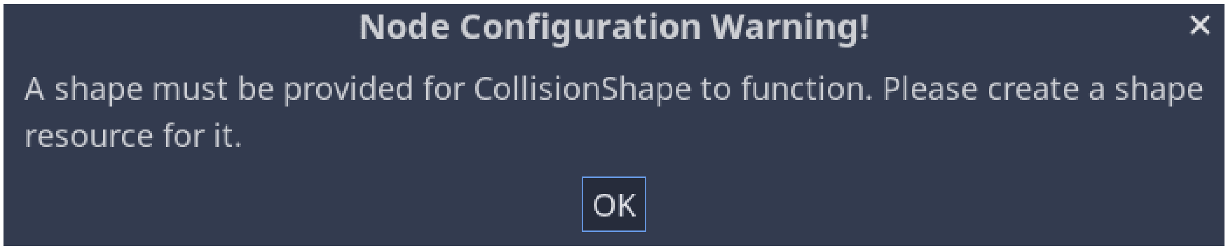

Scene
Import
Filter
Main
main
Favorites:
Search:
staticbod
Matches:
Node
CanvasItem
Node2D
CollisionObject2D
PhysicsBody2D
StaticBody2D
Node3D
CollisionObject3D
PhysicsBody3D
StaticBody3D
Recent:
Node
FileSystem
res://art/
Filter Files
res://
art
body.material
eye.material
House In a Forest Lo
mob.glb
Description:
Cancel
Create

StaticBody3D: Physics body for 3D physics which is static or moves only by script. Useful for floor and walls.

Your scene tree should look like this


Scene
Import
Filter nodes
Main
Ground
CollisionShape

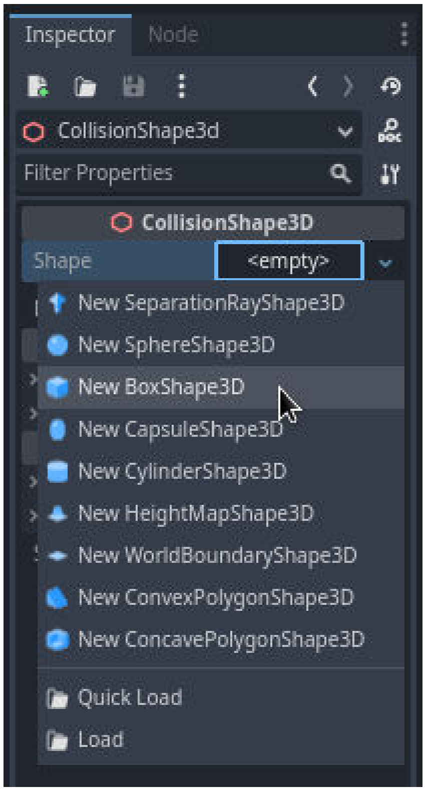
A warning sign next to the CollisionShape3D appears because we haven't defined its shape. If you click the icon, a popup appears to give you more information.



Node Configuration Warning!

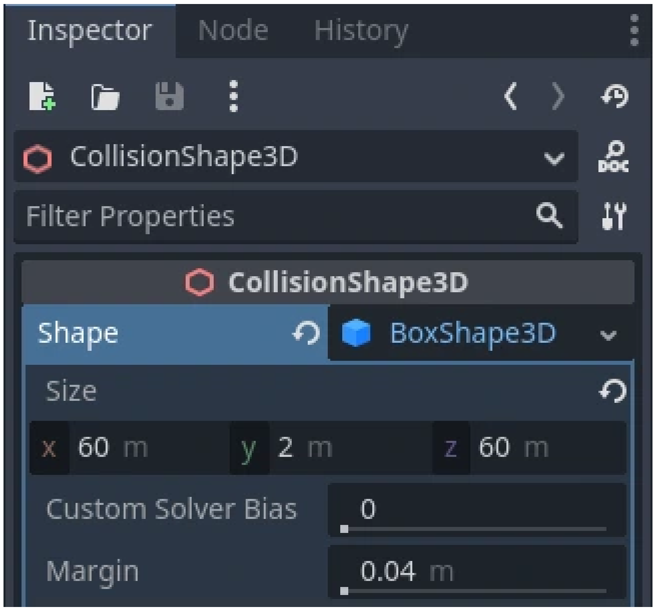
A shape must be provided for CollisionShape to function. Please create a shape resource for it.

To create a shape, select the CollisionShape3D node, head to the *Inspector* and click the *<empty>* field next to the *Shape* property. Create a new *BoxShape3D*.


Inspector
Node
CollisionShape3d
Filter Properties
CollisionShape3D
Shape
<empty>
New SeparationRayShape3D
New SphereShape3D
New BoxShape3D
New CapsuleShape3D
New CylinderShape3D
New HeightMapShape3D
New WorldBoundaryShape3D
New ConvexPolygonShape3D
New ConcavePolygonShape3D
Quick Load
Load

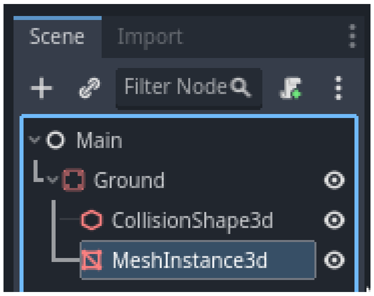
The box shape is perfect for flat ground and walls. Its thickness makes it reliable to block even fast-moving objects.

A box's wireframe appears in the viewport with three orange dots. You can click and drag these to edit the shape's extents interactively. We can also precisely set the size in the inspector. Click on the BoxShape3D to expand the resource. Set its *Size* to `60` on the X-axis, `2` for the Y-axis, and `60` for the Z-axis.

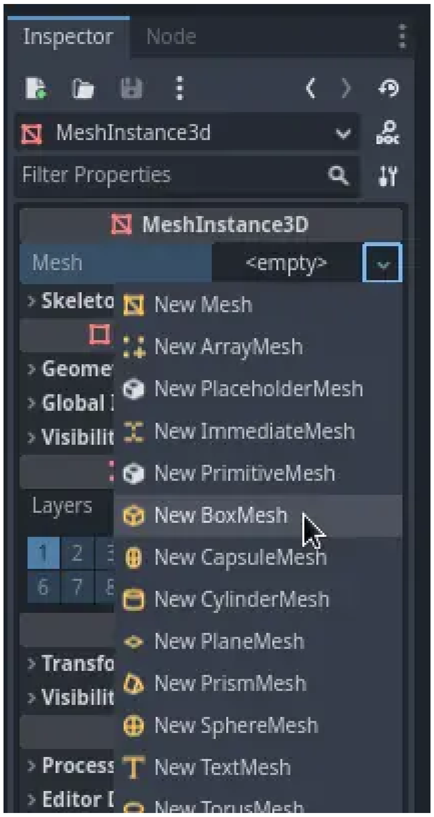

Inspector
Node
History
CollisionShape3D
Filter Properties
CollisionShape3D

Shape: BoxShape3D
Size: x 60 m, y 2 m, z 60 m
Custom Solver Bias: 0
Margin: 0.04 m

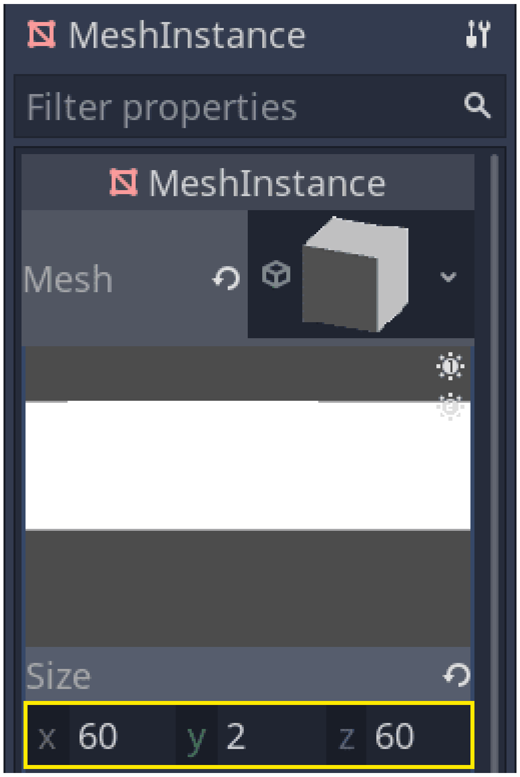
Collision shapes are invisible. We need to add a visual floor that goes along with it. Select the `Ground` node and add a MeshInstance3D as its child.


Scene
Import
Filter Node
Main
Ground
CollisionShape3d
MeshInstance3d

In the *Inspector*, click on the field next to *Mesh* and create a BoxMesh resource to create a visible box.


Inspector
Node
MeshInstance3d
Filter Properties
MeshInstance3D
Mesh
<empty>
Skeleto
Geome
Global I
Visibilit
Layers
1
2
6
7
Transfo
Visibilit
Process
Editor I
New Mesh
New ArrayMesh
New PlaceholderMesh
New ImmediateMesh
New PrimitiveMesh
New BoxMesh
New CapsuleMesh
New CylinderMesh
New PlaneMesh
New PrismMesh
New SphereMesh
New TextMesh
New TorusMesh

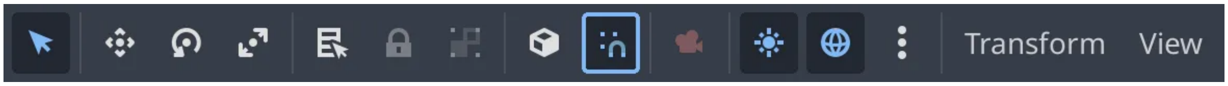
Once again, it's too small by default. Click the box icon to expand the resource and set its *Size* to `60`, `2`, and `60`.


MeshInstance
Filter properties
MeshInstance
Mesh
Size

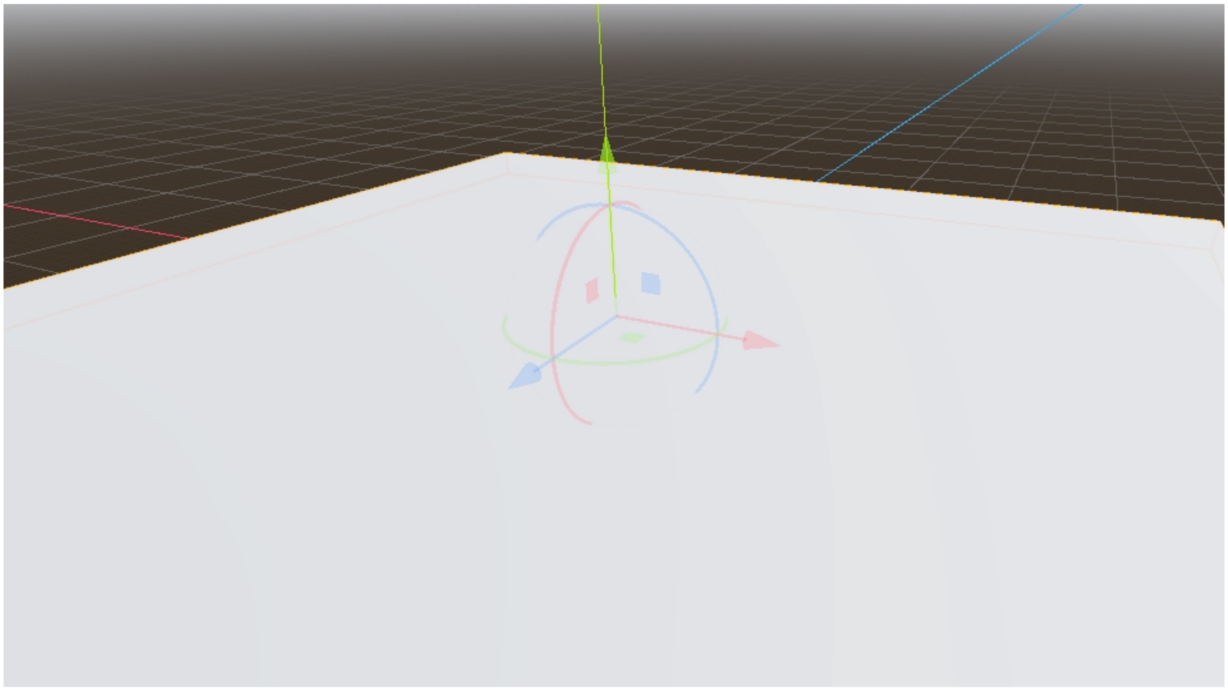
x 60 y 2 z 60

You should see a wide grey slab that covers the grid and blue and red axes in the viewport.

We're going to move the ground down so we can see the floor grid. To do this, the grid snapping feature can be used. Grid snapping can be activated 2 ways in the 3D editor. The first is by pressing the *Use Snap* button (or pressing the `Y` key). The second is by selecting a node, dragging a handle on the gizmo **then** holding `Ctrl` while still clicking to enable snapping as long as `Ctrl` is held.

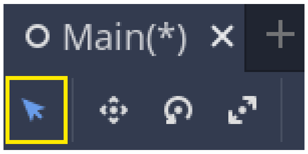

Transform
View

Start by setting snapping with your preferred method. Then move the `Ground` node using the Y-axis (the green arrow on the gizmo).

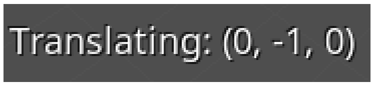


Note

If you can't see the 3D object manipulator like on the image above, ensure the *Select Mode* is active in the toolbar above the view.



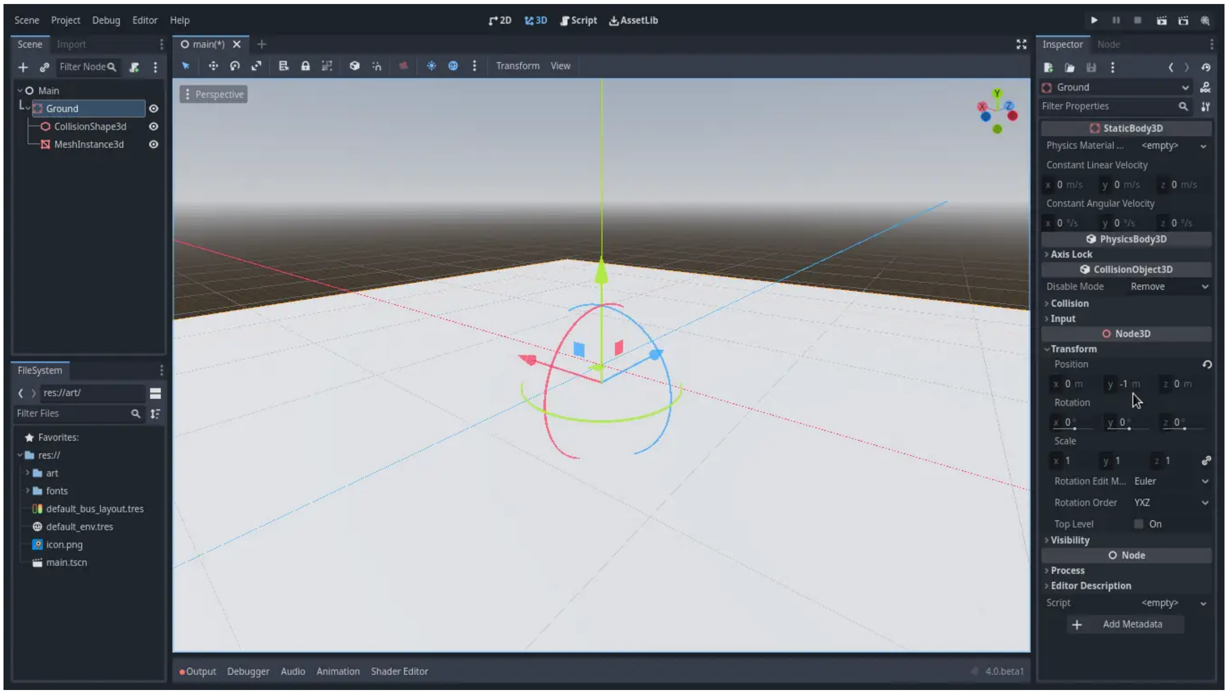
Move the ground down `1` meter, in order to have a visible editor grid. A label in the bottom-left corner of the viewport tells you how much you're translating the node.


Translating: (0, -1, 0)

Note

Moving the *Ground* node down moves both children along with it. Ensure you move the *Ground* node, **not** the *MeshInstance3D* or the *CollisionShape3D*.

Ultimately, `Ground`'s transform.position.y should be -1



Ground

StaticBody3D
Physics Material ... <empty>
Constant Linear Velocity x 0 m/s y 0 m/s z 0 m/s
Constant Angular Velocity x 0 °/s y 0 °/s z 0 °/s

PhysicsBody3D
Axis Lock

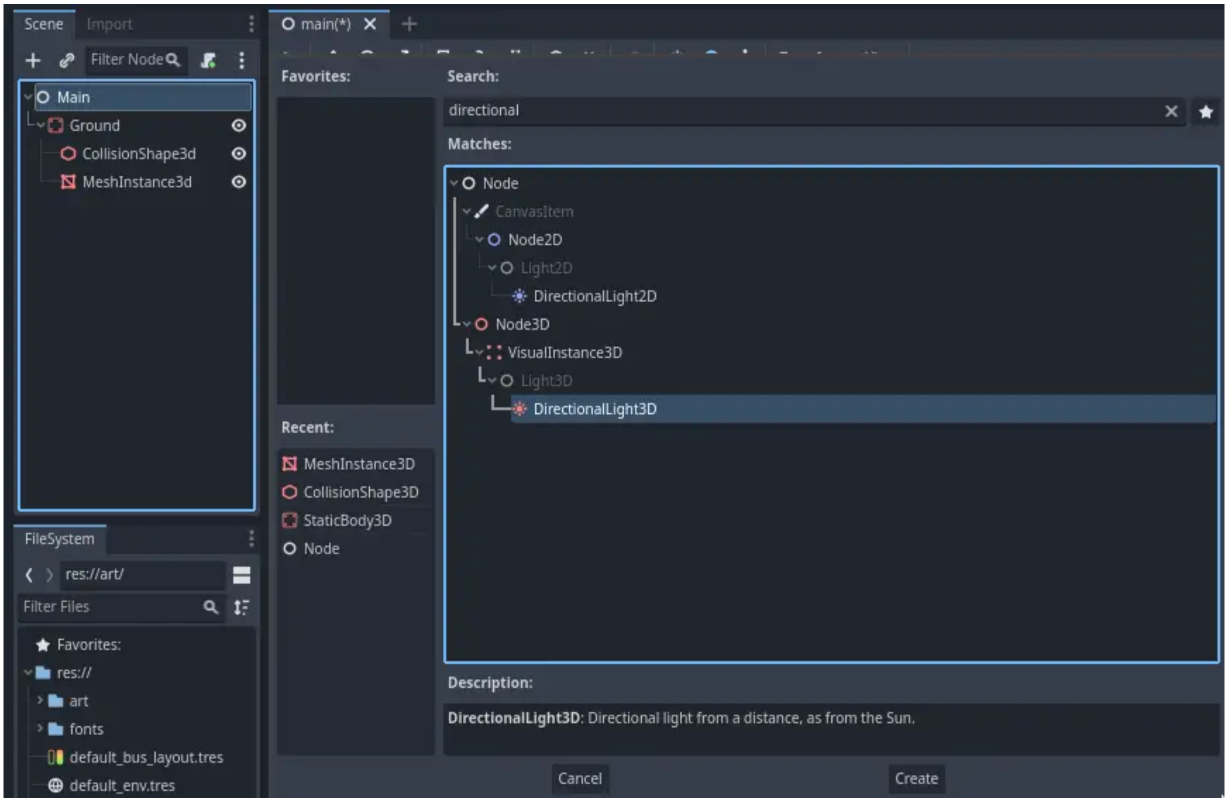
CollisionObject3D
Disable Mode Remove
Collision
Input

Node3D
Transform
Position x 0 m y -1 m z 0 m
Rotation x 0° y 0° z 0°
Scale x 1 y 1 z 1
Rotation Edit M... Euler
Rotation Order YXZ
Top Level ☐ On
Visibility

Node
Process
Editor Description
Script <empty>

Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Scene
Import
Filter Nodes
Main
Ground
CollisionShape3d
MeshInstance3d
main(*)
Transform
View
Perspective
FileSystem
res://art/
Filter Files
Favorites:
res://
art
fonts
default_bus_layout.tres
default_env.tres
icon.png
main.tscn
Output
Debugger
Audio
Animation
Shader Editor
4.0.beta1
Inspector
Node
Filter Properties
Add Metadata

Let's add a directional light so our scene isn't all grey. Select the `Main` node and add a child node DirectionalLight3D.

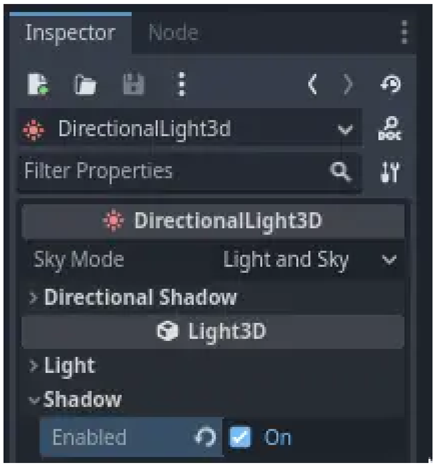

Scene
Import
Filter Nodes
Main
Ground
CollisionShape3d
MeshInstance3d
FileSystem
res://art/
Filter Files
Favorites:
res://
art
fonts
default_bus_layout.tres
default_env.tres
main(*)
Favorites:
Recent:
MeshInstance3D
CollisionShape3D
StaticBody3D
Node
Search:
Matches:
Node
CanvasItem
Node2D
Light2D
DirectionalLight2D
Node3D
VisualInstance3D
Light3D
DirectionalLight3D
Description:
Cancel
Create

Search: directional

DirectionalLight3D: Directional light from a distance, as from the Sun.

We need to move and rotate the DirectionalLight3D node. Move it up by clicking and dragging on the manipulator's green arrow and click and drag on the red arc to rotate it around the X-axis, until the ground is lit.

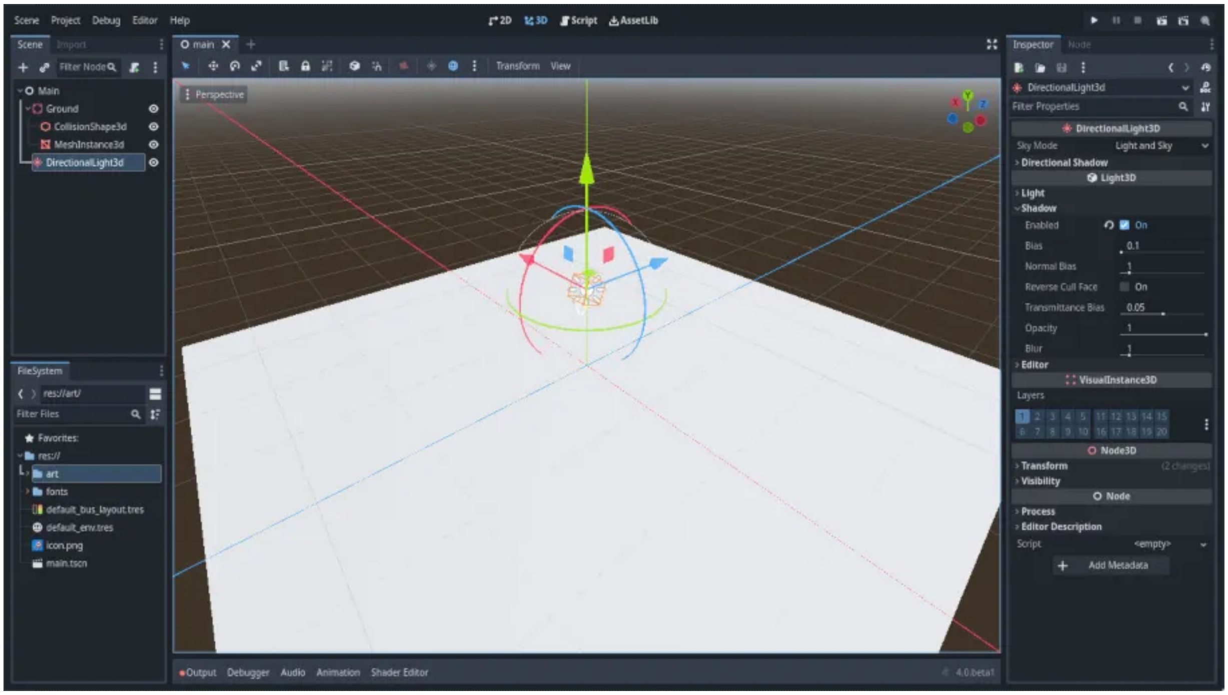
In the *Inspector*, turn on *Shadow -> Enabled* by clicking the checkbox.


Inspector
Node
DirectionalLight3d
Filter Properties
DirectionalLight3D
Directional Shadow
Light3D
Light
Shadow

Sky Mode: Light and Sky

Enabled ☑ On

At this point, your project should look like this.


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Scene
Import
Filter Nodes
Main
Ground
CollisionShape3d
MeshInstance3d
DirectionalLight3d
main
Transform
View
Perspective
FileSystem
res://art/
Filter Files
Favorites:
res://
art
fonts
default_bus_layout.tres
default_env.tres
icon.png
main.tscn
Output
Debugger
Audio
Animation
Shader Editor
4.0.beta1
Inspector
Node
DirectionalLight3d
Filter Properties
DirectionalLight3D
Directional Shadow
Light3D
Light
Shadow
Editor
VisualInstance3D
Layers
Node3D
Transform
(2 changes)
Visibility
Node
Process
Editor Description
Add Metadata

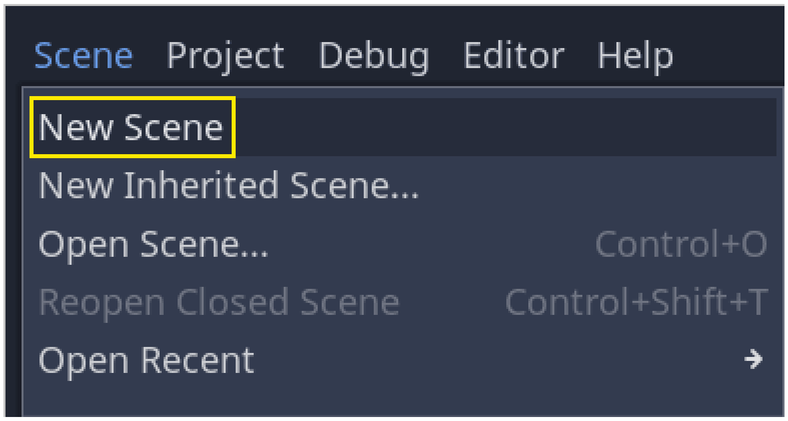
Sky Mode: Light and Sky
Enabled: On
Bias: 0.1
Normal Bias: 1
Reverse Cull Face: On
Transmittance Bias: 0.05
Opacity: 1
Blur: 1
Script: <empty>

That's our starting point. In the next part, we will work on the player scene and base movement.

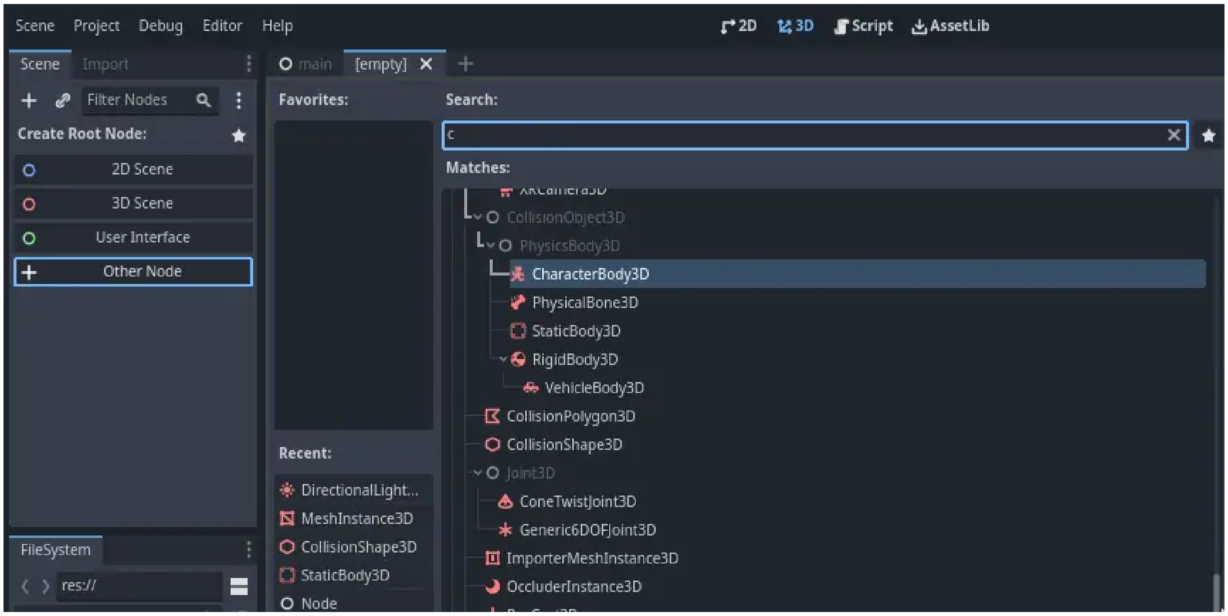
Player scene and input actions

In the next two lessons, we will design the player scene, register custom input actions, and code player movement. By the end, you'll have a playable character that moves in eight directions.

Create a new scene by going to the Scene menu in the top-left and clicking *New Scene*.


Scene
Project
Debug
Editor
Help
New Scene
New Inherited Scene...
Open Scene... Control+O
Reopen Closed Scene Control+Shift+T
Open Recent

Create a CharacterBody3D node as the root


Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Scene
Import
main
[empty]
Filter Nodes
Create Root Node:
2D Scene
3D Scene
User Interface
Other Node
Favorites:
Search:
c
Matches:
CollisionObject3D
PhysicsBody3D
CharacterBody3D
PhysicalBone3D
StaticBody3D
RigidBody3D
VehicleBody3D
CollisionPolygon3D
CollisionShape3D
Joint3D
ConeTwistJoint3D
Generic6DOFJoint3D
ImporterMeshInstance3D
OccluderInstance3D
Recent:
DirectionalLight...
MeshInstance3D
CollisionShape3D
StaticBody3D
Node
FileSystem
res://

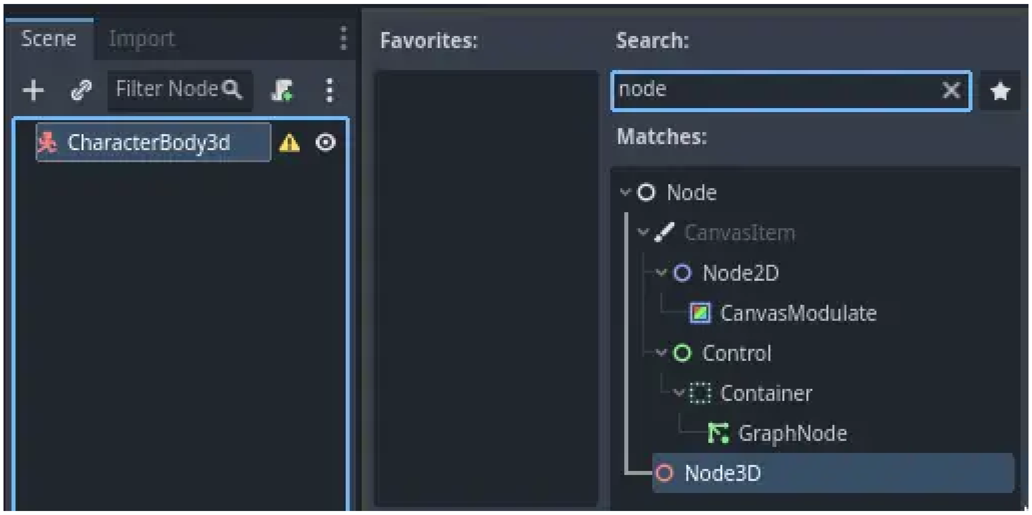
Name the CharacterBody3D to `Player`. Character bodies are complementary to the area and rigid bodies used in the 2D game tutorial. Like rigid bodies, they can move and collide with the environment, but instead of being controlled by the physics engine, **you** dictate their movement. You will see how we use the node's unique features when we code the jump and squash mechanics.

See also

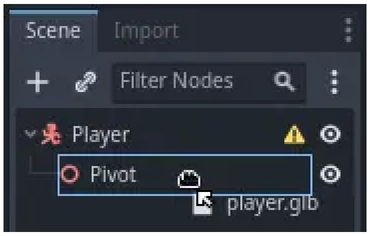
To learn more about the different physics node types, see the Physics introduction.

For now, we're going to create a basic rig for our character's 3D model. This will allow us to rotate the model later via code while it plays an animation.

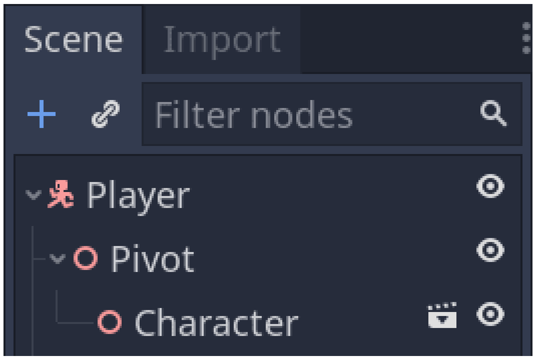
Add a Node3D node as a child of `Player` and name it `Pivot`


Scene
Import
Filter Nodes
CharacterBody3d
Favorites:
Search:
node
Matches:
Node
CanvasItem
Node2D
CanvasModulate
Control
Container
GraphNode
Node3D

Then, in the FileSystem dock, expand the `art/` folder by double-clicking it and drag and drop `player.glb` onto `Pivot`.


Scene
Import
Filter Nodes
Player
Pivot
player.glb

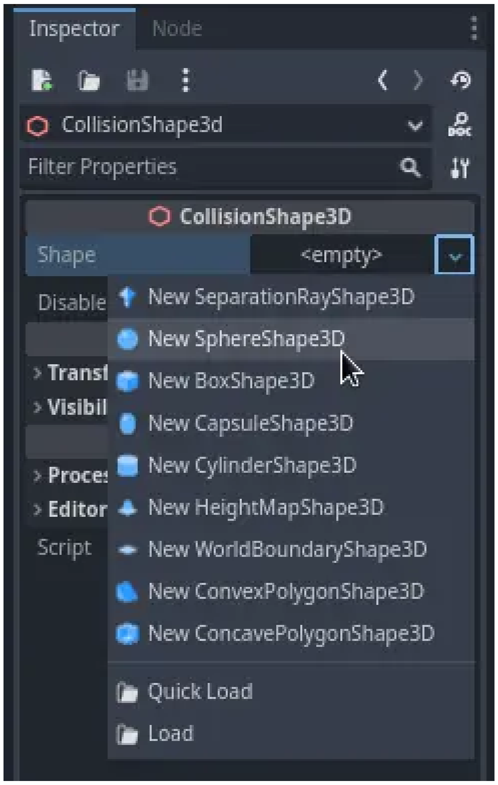
This should instantiate the model as a child of `Pivot`. You can rename it to `Character`.


Scene
Import
Filter nodes
Player
Pivot
Character

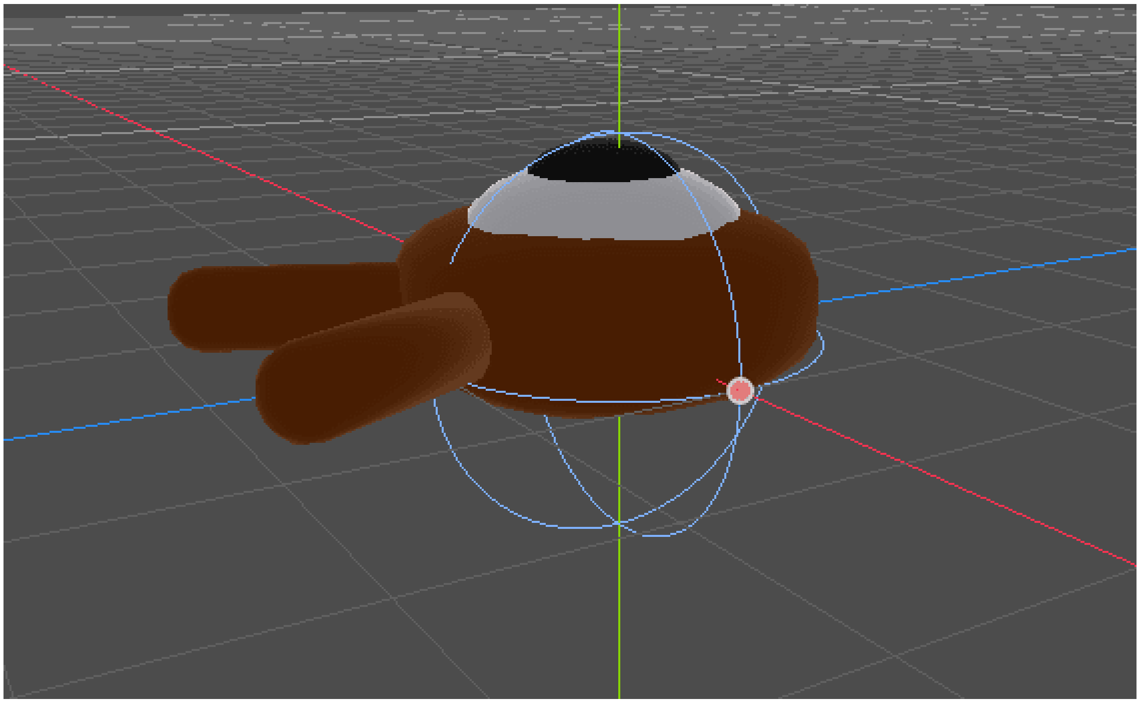
Note

The `.glb` files contain 3D scene data based on the open source glTF 2.0 specification. They're a modern and powerful alternative to a proprietary format like FBX, which Godot also supports. To produce these files, we designed the model in Blender 3D [https://www.blender.org/] and exported it to glTF.

As with all kinds of physics nodes, we need a collision shape for our character to collide with the environment. Select the `Player` node again and add a child node CollisionShape3D. In the *Inspector*, on the *Shape* property, add a new SphereShape3D.

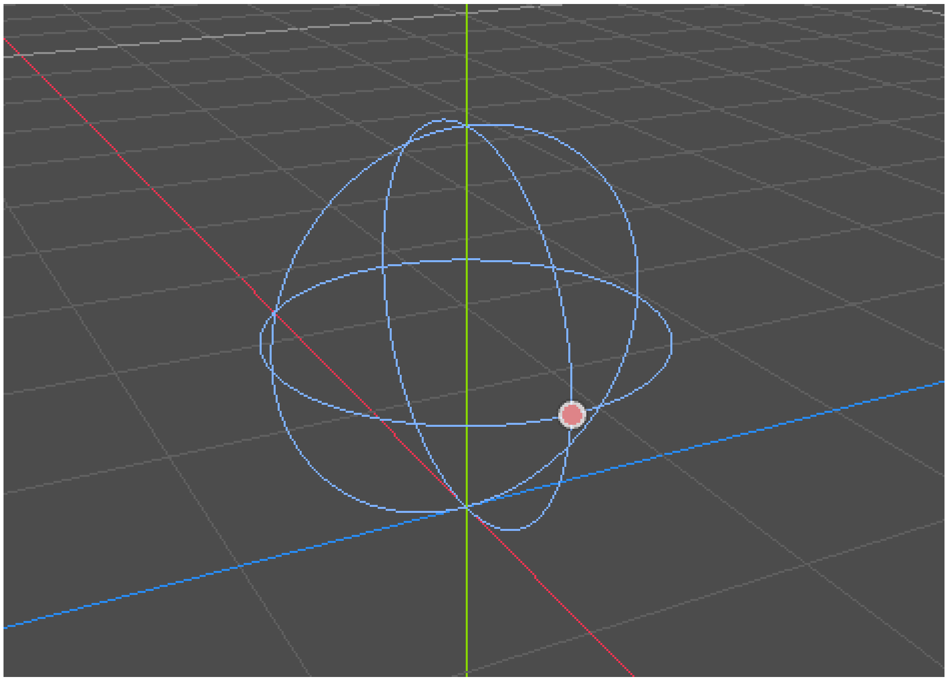

Inspector
Node
CollisionShape3d
Filter Properties
CollisionShape3D
Shape
<empty>
Disable
Transf
Visibil
Proces
Editor
Script
New SeparationRayShape3D
New SphereShape3D
New BoxShape3D
New CapsuleShape3D
New CylinderShape3D
New HeightMapShape3D
New WorldBoundaryShape3D
New ConvexPolygonShape3D
New ConcavePolygonShape3D
Quick Load
Load

The sphere's wireframe appears below the character.

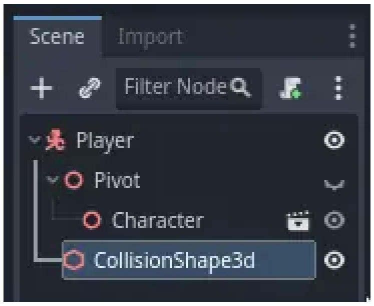


It will be the shape the physics engine uses to collide with the environment, so we want it to better fit the 3D model. Make it a bit larger by dragging the orange dot in the viewport. My sphere has a radius of about `0.8` meters.

Then, move the collision shape up so its bottom roughly aligns with the grid's plane.



To make moving the shape easier, you can toggle the model's visibility by clicking the eye icon next to the `Character` or the `Pivot` nodes.


Scene
Import
Filter Node
Player
Pivot
Character
CollisionShape3d

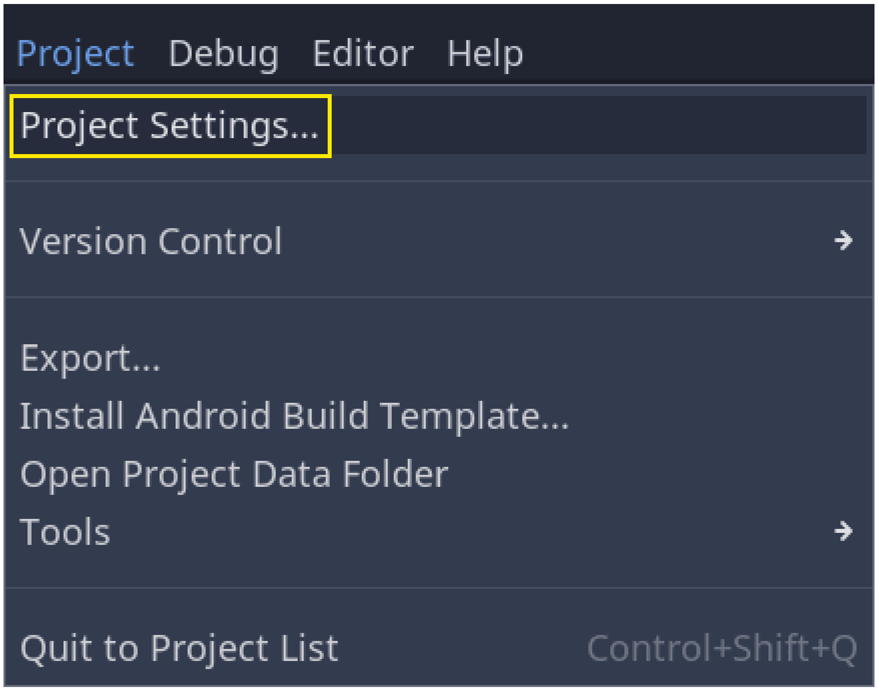
Save the scene as `player.tscn`

With the nodes ready, we can almost get coding. But first, we need to define some input actions.

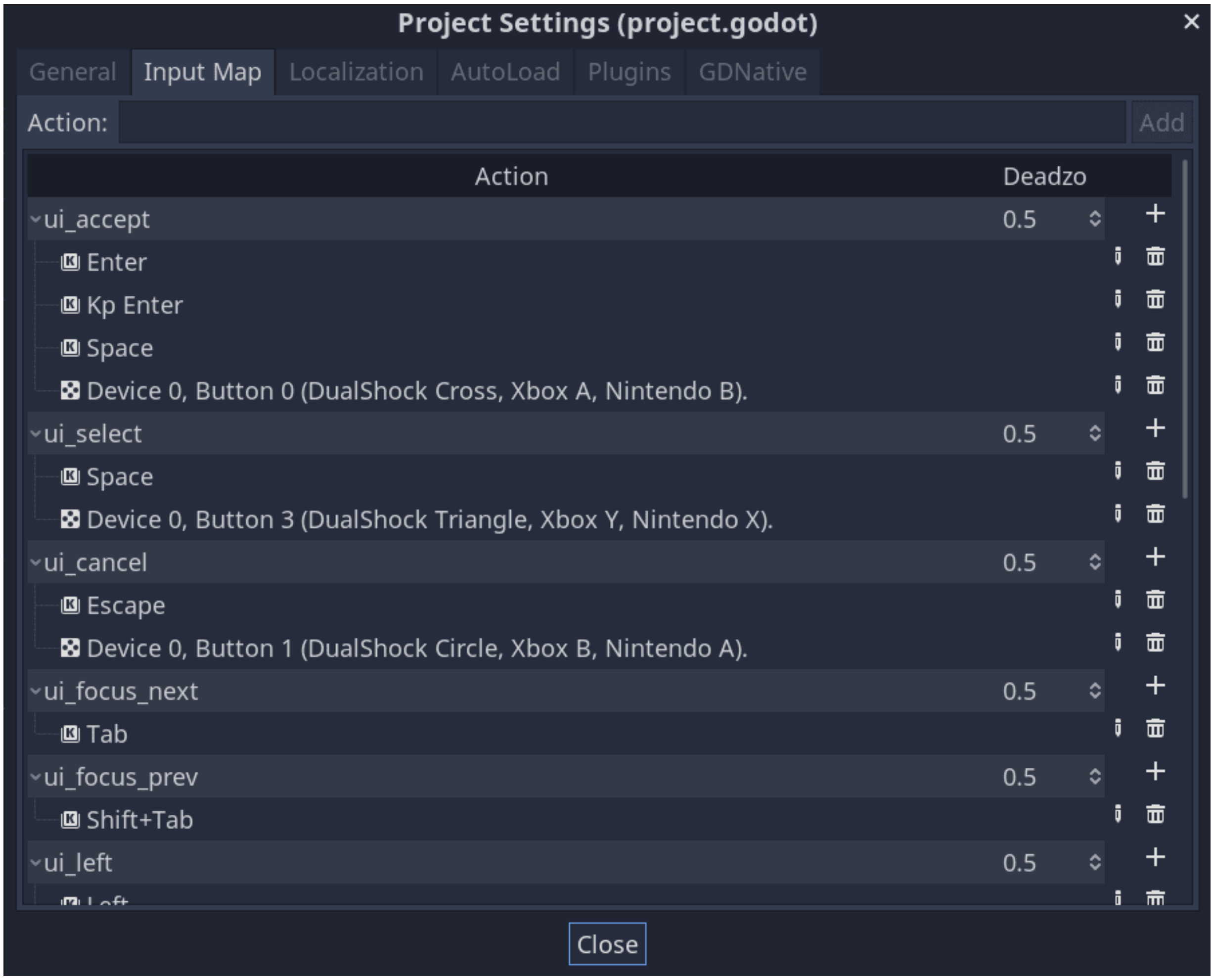
Creating input actions

To move the character, we will listen to the player's input, like pressing the arrow keys. In Godot, while we could write all the key bindings in code, there's a powerful system that allows you to assign a label to a set of keys and buttons. This simplifies our scripts and makes them more readable.

This system is the Input Map. To access its editor, head to the *Project* menu and select *Project Settings*.


Project
Debug
Editor
Help
Project Settings...
Version Control
Export...
Install Android Build Template...
Open Project Data Folder
Tools
Quit to Project List
Control+Shift+Q

At the top, there are multiple tabs. Click on *Input Map*. This window allows you to add new actions at the top; they are your labels. In the bottom part, you can bind keys to these actions.



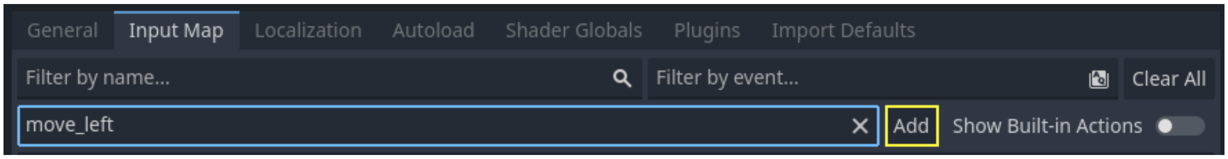
Project Settings (project.godot)
General
Input Map
Localization
AutoLoad
Plugins
GDNative
Action:
Add
Close

Action	Deadzo
ui_accept	0.5
Enter	
Kp Enter	
Space	
Device 0, Button 0 (DualShock Cross, Xbox A, Nintendo B).	
ui_select	0.5
Space	
Device 0, Button 3 (DualShock Triangle, Xbox Y, Nintendo X).	
ui_cancel	0.5
Escape	
Device 0, Button 1 (DualShock Circle, Xbox B, Nintendo A).	
ui_focus_next	0.5
Tab	
ui_focus_prev	0.5
Shift+Tab	
ui_left	0.5

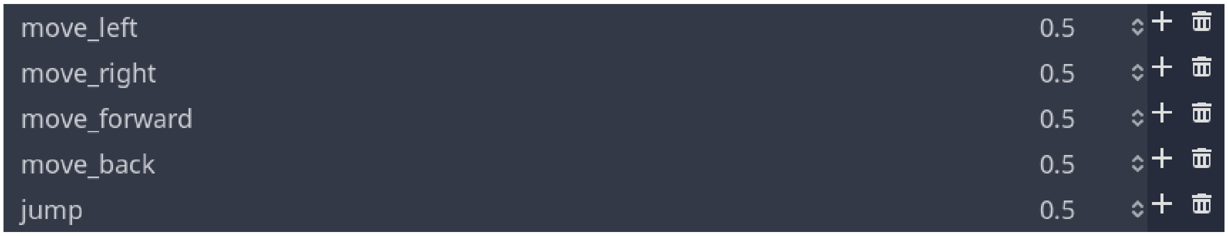
Godot projects come with some predefined actions designed for user interface design (see above screenshot). These will become visible if you enable the *Show Built-in Actions* toggle. We could use these here, but instead we're defining our own to support gamepads. Leave *Show Built-in Actions* disabled.

We're going to name our actions `move_left`, `move_right`, `move_forward`, `move_back`, and `jump`.

To add an action, write its name in the bar at the top and press Enter.

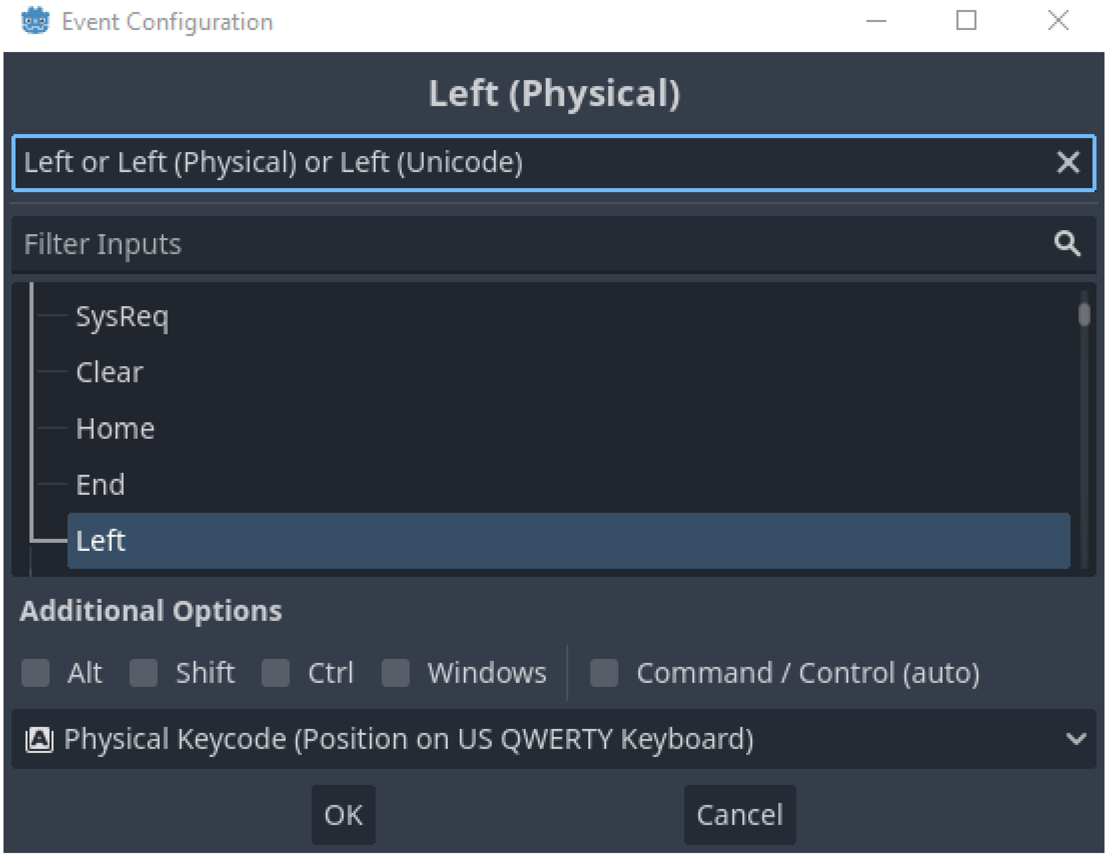

General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Filter by name...
Filter by event...
Clear All
move_left
Add
Show Built-in Actions

Create the following five actions:

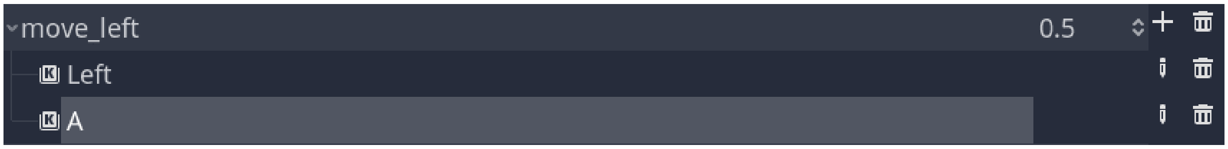


move_left	0.5
move_right	0.5
move_forward	0.5
move_back	0.5
jump	0.5

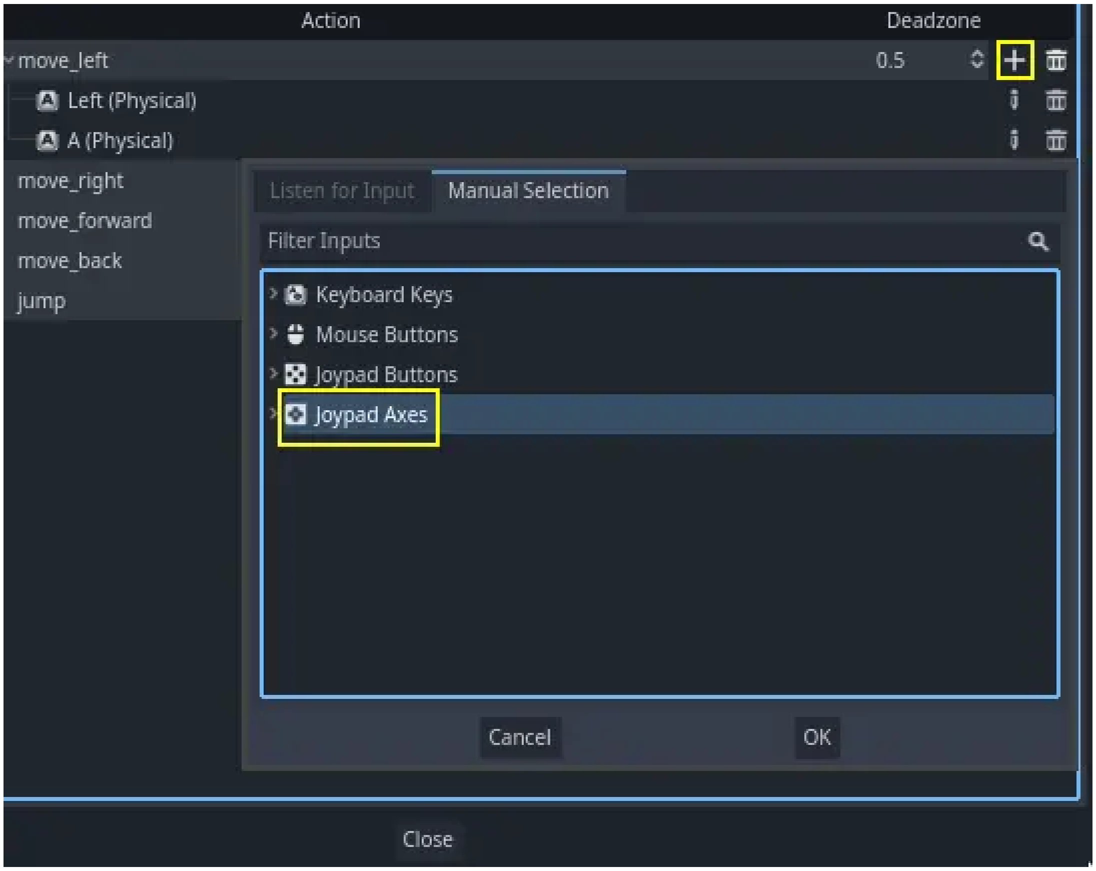
To bind a key or button to an action, click the "+" button to its right. Do this for `move_left`. Press the left arrow key and click *OK*.


Event Configuration
Left (Physical)
Left or Left (Physical) or Left (Unicode)
Filter Inputs
SysReq
Clear
Home
End
Left
Additional Options
Alt
Shift
Ctrl
Windows
Command / Control (auto)
Physical Keycode (Position on US QWERTY Keyboard)
OK
Cancel

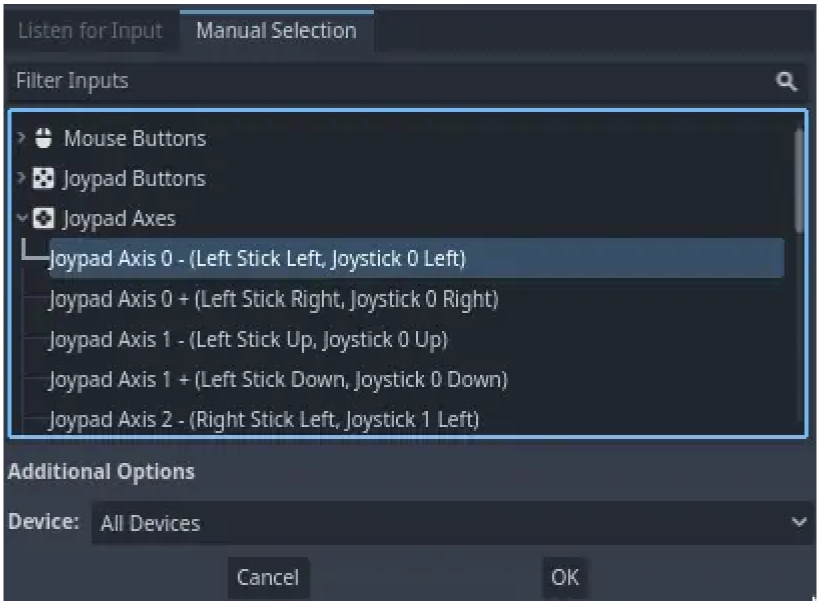
Bind also the `A` key, onto the action `move_left`.


move_left
0.5
Left
A

Let's now add support for a gamepad's left joystick. Click the "+" button again but this time, select *Manual Selection -> Joypad Axes*.


Action
Deadzone
move_left
0.5
Left (Physical)
A (Physical)
move_right
move_forward
move_back
jump
Listen for Input
Manual Selection
Filter Inputs
Keyboard Keys
Mouse Buttons
Joypad Buttons
Joypad Axes
Cancel
OK
Close

Select the negative X axis of the left joystick.


Listen for Input
Manual Selection
Filter Inputs
Mouse Buttons
Joypad Buttons
Joypad Axes
Joypad Axis 0 - (Left Stick Left, Joystick 0 Left)
Joypad Axis 0 + (Left Stick Right, Joystick 0 Right)
Joypad Axis 1 - (Left Stick Up, Joystick 0 Up)
Joypad Axis 1 + (Left Stick Down, Joystick 0 Down)
Joypad Axis 2 - (Right Stick Left, Joystick 1 Left)
Additional Options
Cancel
OK

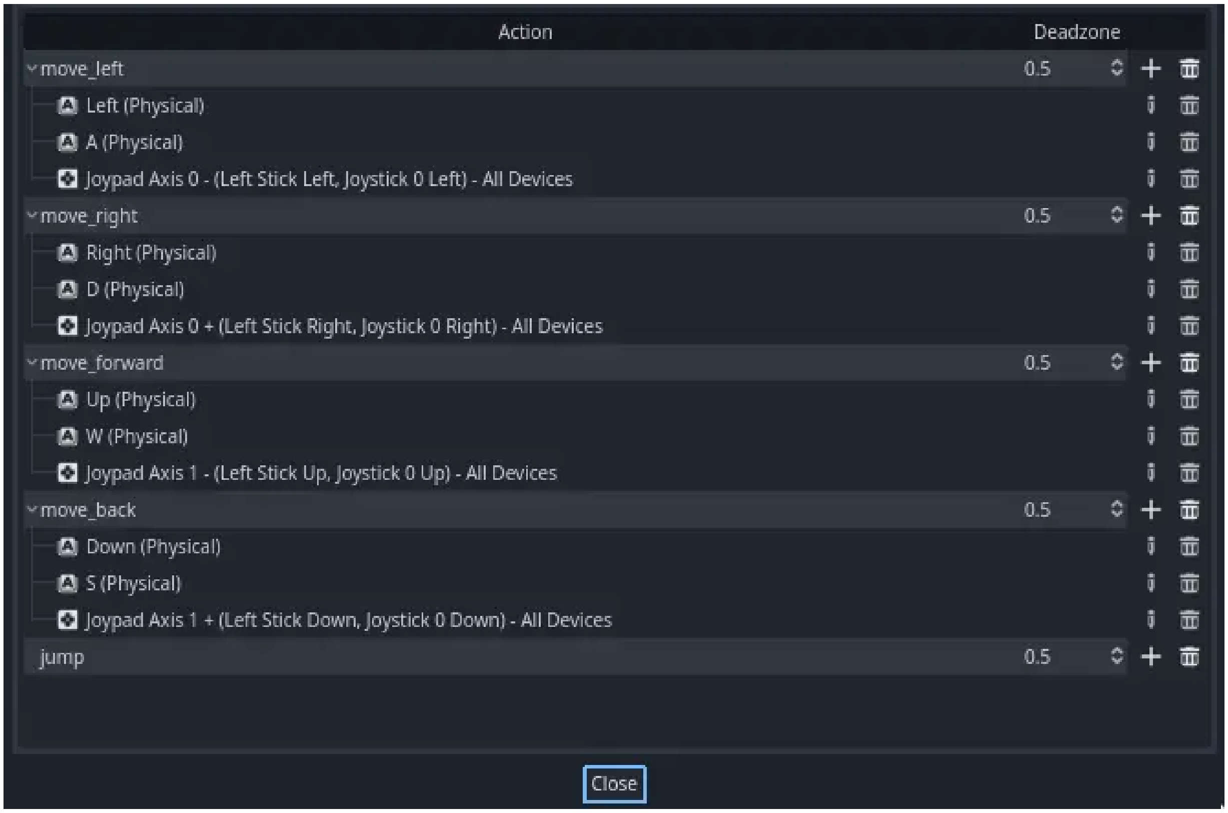
Device: All Devices

Leave the other values as default and press *OK*

Note

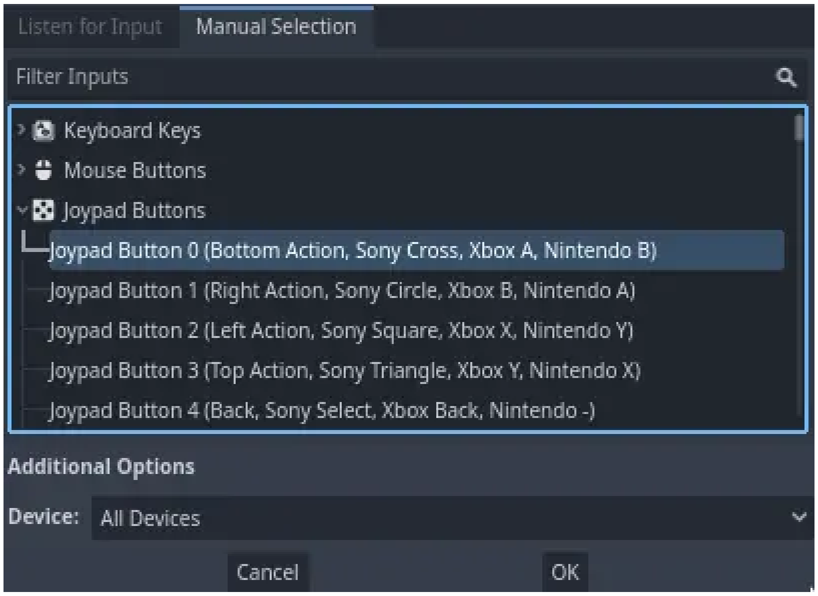
If you want controllers to have different input actions, you should use the Devices option in Additional Options. Device 0 corresponds to the first plugged gamepad, Device 1 corresponds to the second plugged gamepad, and so on.

Do the same for the other input actions. For example, bind the right arrow, D, and the left joystick's positive axis to `move_right`. After binding all keys, your interface should look like this.



Action	Deadzone
move_left	0.5
Left (Physical)	
A (Physical)	
Joypad Axis 0 - (Left Stick Left, Joystick 0 Left) - All Devices	
move_right	0.5
Right (Physical)	
D (Physical)	
Joypad Axis 0 + (Left Stick Right, Joystick 0 Right) - All Devices	
move_forward	0.5
Up (Physical)	
W (Physical)	
Joypad Axis 1 - (Left Stick Up, Joystick 0 Up) - All Devices	
move_back	0.5
Down (Physical)	
S (Physical)	
Joypad Axis 1 + (Left Stick Down, Joystick 0 Down) - All Devices	
jump	0.5

The final action to set up is the `jump` action. Bind the Space key and the gamepad's A button.



Listen for Input
Manual Selection
Filter Inputs
Keyboard Keys
Mouse Buttons
Joypad Buttons

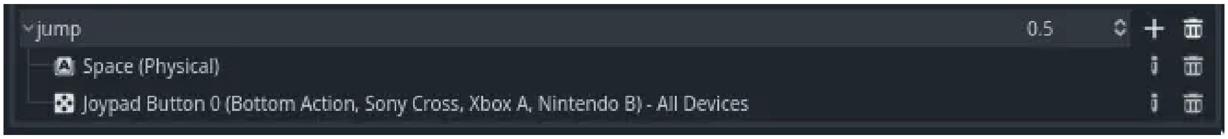
Joypad Button 0 (Bottom Action, Sony Cross, Xbox A, Nintendo B)
Joypad Button 1 (Right Action, Sony Circle, Xbox B, Nintendo A)
Joypad Button 2 (Left Action, Sony Square, Xbox X, Nintendo Y)
Joypad Button 3 (Top Action, Sony Triangle, Xbox Y, Nintendo X)
Joypad Button 4 (Back, Sony Select, Xbox Back, Nintendo -)

Additional Options

Device: All Devices

Cancel
OK

Your jump input action should look like this.



jump 0.5

Space (Physical)

Joypad Button 0 (Bottom Action, Sony Cross, Xbox A, Nintendo B) - All Devices

That's all the actions we need for this game. You can use this menu to label any groups of keys and buttons in your projects.

In the next part, we'll code and test the player's movement.

Moving the player with code

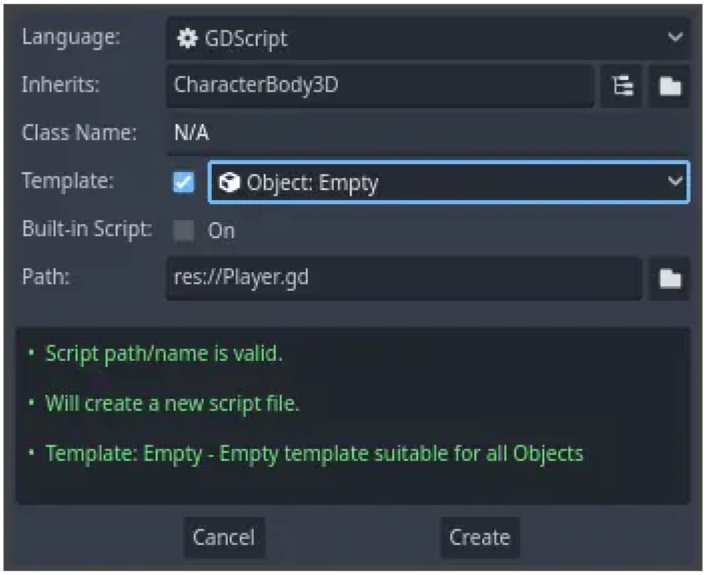
It's time to code! We're going to use the input actions we created in the last part to move the character.

Note

For this project, we will be following the Godot naming conventions.

- **GDScript**: Classes (nodes) use PascalCase, variables and functions use snake_case, and constants use ALL_CAPS (See GDScript style guide).
- **C#**: Classes, export variables and methods use PascalCase, private fields use _camelCase, local variables and parameters use camelCase (See C# style guide). Be careful to type the method names precisely when connecting signals.

Right-click the `Player` node and select *Attach Script* to add a new script to it. In the popup, set the *Template* to *Empty* before pressing the *Create* button. We set it to *Empty* because we want to write our own code for player movement.


Language: GDScript

Inherits: CharacterBody3D

Class Name: N/A

Template: ☑ Object: Empty

Built-in Script: ☐ On

Path: res://Player.gd

- Script path/name is valid.
- Will create a new script file.
- Template: Empty - Empty template suitable for all Objects

Let's start with the class's properties. We're going to define a movement speed, a fall acceleration representing gravity, and a velocity we'll use to move the character.

GDScript

```
extends CharacterBody3D

# How fast the player moves in meters per second.
@export var speed = 14
# The downward acceleration when in the air, in meters per second squared.
@export var fall_acceleration = 75

var target_velocity = Vector3.ZERO
```

C#

```
using Godot;

public partial class Player : CharacterBody3D
{
    // Don't forget to rebuild the project so the editor knows about the new export variable.

    // How fast the player moves in meters per second.
    [Export]
    public int Speed { get; set; } = 14;
    // The downward acceleration when in the air, in meters per
second squared.
    [Export]
    public int FallAcceleration { get; set; } = 75;

    private Vector3 _targetVelocity = Vector3.Zero;
}
```

These are common properties for a moving body. The `target_velocity` is a 3D vector combining a speed with a direction. Here, we define it as a property because we want to update and reuse its value across frames.

Note

The values are quite different from 2D code because distances are in meters. While in 2D, a thousand units (pixels) may only correspond to half of your screen's width, in 3D, it's a kilometer.

Let's code the movement. We start by calculating the input direction vector using the global `Input` object, in `_physics_process()`.

GDScript

```
func _physics_process(delta):
    # We create a local variable to store the input direction.
    var direction = Vector3.ZERO

    # We check for each move input and update the direction
accordingly.
    if Input.is_action_pressed("move_right"):
        direction.x += 1
    if Input.is_action_pressed("move_left"):
        direction.x -= 1
    if Input.is_action_pressed("move_back"):
        # Notice how we are working with the vector's x and z
```

```
axes.
        # In 3D, the XZ plane is the ground plane.
        direction.z += 1
    if Input.is_action_pressed("move_forward"):
        direction.z -= 1
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // We create a local variable to store the input direction.
    var direction = Vector3.Zero;

    // We check for each move input and update the direction
accordingly.
    if (Input.IsActionPressed("move_right"))
    {
        direction.X += 1.0f;
    }
    if (Input.IsActionPressed("move_left"))
    {
        direction.X -= 1.0f;
    }
    if (Input.IsActionPressed("move_back"))
    {
        // Notice how we are working with the vector's X and Z
axes.
        // In 3D, the XZ plane is the ground plane.
        direction.Z += 1.0f;
    }
    if (Input.IsActionPressed("move_forward"))
    {
        direction.Z -= 1.0f;
    }
}
```

Here, we're going to make all calculations using the `_physics_process()` virtual function. Like `_process()`, it allows you to update the node every frame, but it's designed specifically for physics-related code like moving a kinematic or rigid body.

See also

To learn more about the difference between `_process()` and `_physics_process()`, see Idle and Physics Processing.

We start by initializing a `direction` variable to `Vector3.ZERO`. Then, we check if the player is pressing one or more of the `move_*` inputs and update the vector's `x` and `z` components accordingly. These correspond to the ground plane's axes.

These four conditions give us eight possibilities and eight possible directions.

In case the player presses, say, both W and D simultaneously, the vector will have a length of about `1.4`. But if they press a single key, it will have a length of `1`. We want the vector's length to be consistent, and not move faster diagonally. To do so, we can call its `normalized()` method.

GDScript

```
#func _physics_process(delta):
    #...

    if direction != Vector3.ZERO:
        direction = direction.normalized()
        # Setting the basis property will affect the rotation of the node.
        $Pivot.basis = Basis.looking_at(direction)
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...

    if (direction != Vector3.Zero)
    {
        direction = direction.Normalized();
        // Setting the basis property will affect the rotation
of the node.
        GetNode<Node3D>("Pivot").Basis =
Basis.LookingAt(direction);
    }
}
```

Here, we only normalize the vector if the direction has a length greater than zero, which means the player is pressing a direction key.

We compute the direction the `$Pivot` is looking by creating a Basis that looks in the `direction` direction.

Then, we update the velocity. We have to calculate the ground velocity and the fall speed separately. Be sure to go back one tab so the lines are inside the `_physics_process()` function but outside the condition we just wrote above.

GDScript

```
func _physics_process(delta):
    #...
    if direction != Vector3.ZERO:
        #...

    # Ground Velocity
    target_velocity.x = direction.x * speed
    target_velocity.z = direction.z * speed

    # Vertical Velocity
    if not is_on_floor(): # If in the air, fall towards the
floor. Literally gravity
        target_velocity.y = target_velocity.y -
(fall_acceleration * delta)

    # Moving the Character
    velocity = target_velocity
    move_and_slide()
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...
    if (direction != Vector3.Zero)
    {
        // ...
    }

    // Ground velocity
    _targetVelocity.X = direction.X * Speed;
    _targetVelocity.Z = direction.Z * Speed;

    // Vertical velocity
    if (!IsOnFloor()) // If in the air, fall towards the floor.
Literally gravity
    {
        _targetVelocity.Y -= FallAcceleration * (float)delta;
    }

    // Moving the character
    Velocity = _targetVelocity;
    MoveAndSlide();
}
```

The `CharacterBody3D.is_on_floor()` function returns `true` if the body collided with the floor in this frame. That's why we apply gravity to the `Player` only while it is in the air.

For the vertical velocity, we subtract the fall acceleration multiplied by the delta time every frame. This line of code will cause our character to fall in every frame, as long as it is not on or colliding with the floor.

The physics engine can only detect interactions with walls, the floor, or other bodies during a given frame if movement and collisions happen. We will use this property later to code the jump.

On the last line, we call `CharacterBody3D.move_and_slide()` which is a powerful method of the `CharacterBody3D` class that allows you to move a character smoothly. If it hits a wall midway through a motion, the engine will try to smooth it out for you. It uses the *velocity* value native to the CharacterBody3D

And that's all the code you need to move the character on the floor.

Here is the complete `player.gd` code for reference.

GDScript

```
extends CharacterBody3D

# How fast the player moves in meters per second.
@export var speed = 14
# The downward acceleration when in the air, in meters per second squared.
@export var fall_acceleration = 75

var target_velocity = Vector3.ZERO


func _physics_process(delta):
    var direction = Vector3.ZERO

    if Input.is_action_pressed("move_right"):
        direction.x += 1
    if Input.is_action_pressed("move_left"):
        direction.x -= 1
    if Input.is_action_pressed("move_back"):
        direction.z += 1
    if Input.is_action_pressed("move_forward"):
        direction.z -= 1

    if direction != Vector3.ZERO:
        direction = direction.normalized()
        $Pivot.basis = Basis.looking_at(direction)
    # Ground Velocity
    target_velocity.x = direction.x * speed
    target_velocity.z = direction.z * speed

    # Vertical Velocity
    if not is_on_floor(): # If in the air, fall towards the
floor. Literally gravity
        target_velocity.y = target_velocity.y -
(fall_acceleration * delta)

    # Moving the Character
    velocity = target_velocity
    move_and_slide()
```

C#

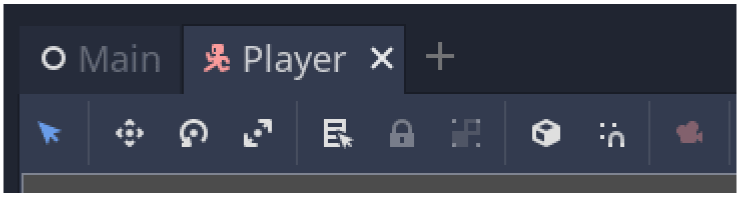
```
using Godot;

public partial class Player : CharacterBody3D
{
    // How fast the player moves in meters per second.
    [Export]
    public int Speed { get; set; } = 14;
    // The downward acceleration when in the air, in meters per
second squared.
    [Export]
    public int FallAcceleration { get; set; } = 75;

    private Vector3 _targetVelocity = Vector3.Zero;

    public override void _PhysicsProcess(double delta)
    {
        var direction = Vector3.Zero;

        if (Input.IsActionPressed("move_right"))
        {
            direction.X += 1.0f;
        }
        if (Input.IsActionPressed("move_left"))
        {
            direction.X -= 1.0f;
        }
        if (Input.IsActionPressed("move_back"))
        {
            direction.Z += 1.0f;
        }
        if (Input.IsActionPressed("move_forward"))
        {
            direction.Z -= 1.0f;
        }

        if (direction != Vector3.Zero)
        {
            direction = direction.Normalized();
            GetNode<Node3D>("Pivot").Basis =
Basis.LookingAt(direction);
        }

        // Ground velocity
        _targetVelocity.X = direction.X * Speed;
        _targetVelocity.Z = direction.Z * Speed;

        // Vertical velocity
        if (!IsOnFloor()) // If in the air, fall towards the
floor. Literally gravity
        {
            _targetVelocity.Y -= FallAcceleration *
(float)delta;
        }

        // Moving the character
        Velocity = _targetVelocity;
        MoveAndSlide();
    }
}
```

Testing our player's movement

We're going to put our player in the `Main` scene to test it. To do so, we need to instantiate the player and then add a camera. Unlike in

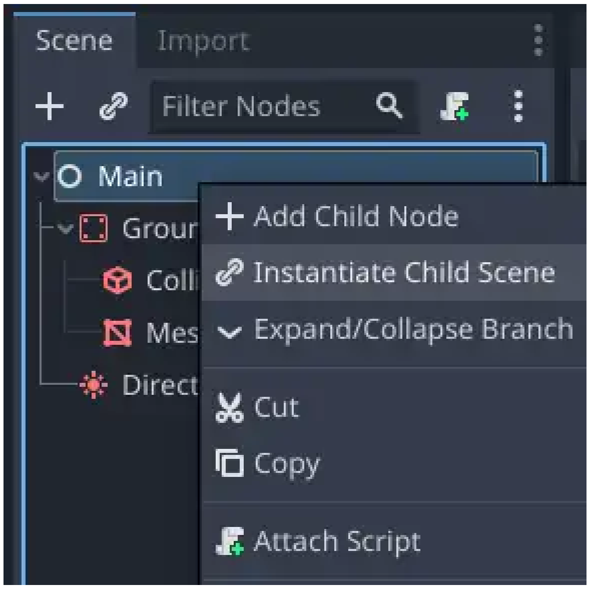
2D, in 3D, you won't see anything if your viewport doesn't have a camera pointing at something.

Save your `Player` scene and open the `Main` scene. You can click on the *Main* tab at the top of the editor to do so.


Main
Player

If you closed the scene before, head to the *FileSystem* dock and double-click `main.tscn` to re-open it.

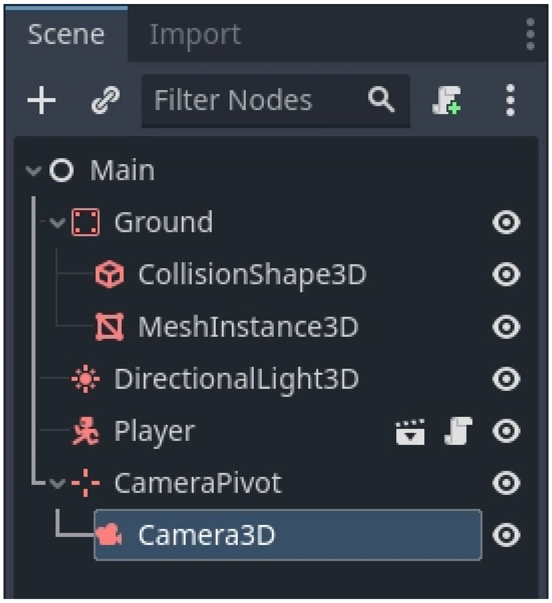
To instantiate the `Player`, right-click on the `Main` node and select *Instantiate Child Scene*.


Scene
Import
Filter Nodes
Main
Grour
Colli
Mes
Direct
Add Child Node
Instantiate Child Scene
Expand/Collapse Branch
Cut
Copy
Attach Script

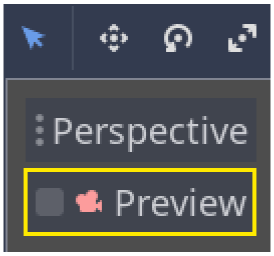
In the popup, double-click `player.tscn`. The character should appear in the center of the viewport.

Adding a camera

Let's add the camera next. Like we did with our *Player*'s *Pivot*, we're going to create a basic rig. Right-click on the `Main` node again and select *Add Child Node*. Create a new Marker3D, and name it `CameraPivot`. Select `CameraPivot` and add a child node Camera3D to it. Your scene tree should look similar to this.

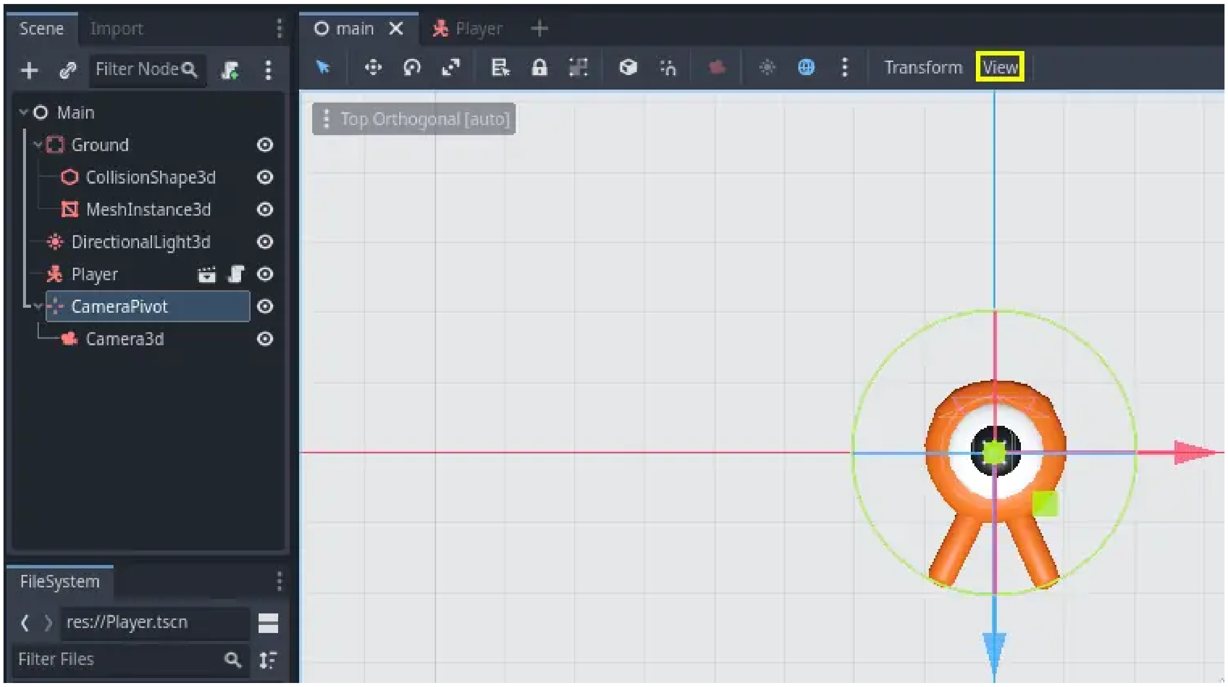

Scene
Import
Filter Nodes
Main
Ground
CollisionShape3D
MeshInstance3D
DirectionalLight3D
Player
CameraPivot
Camera3D

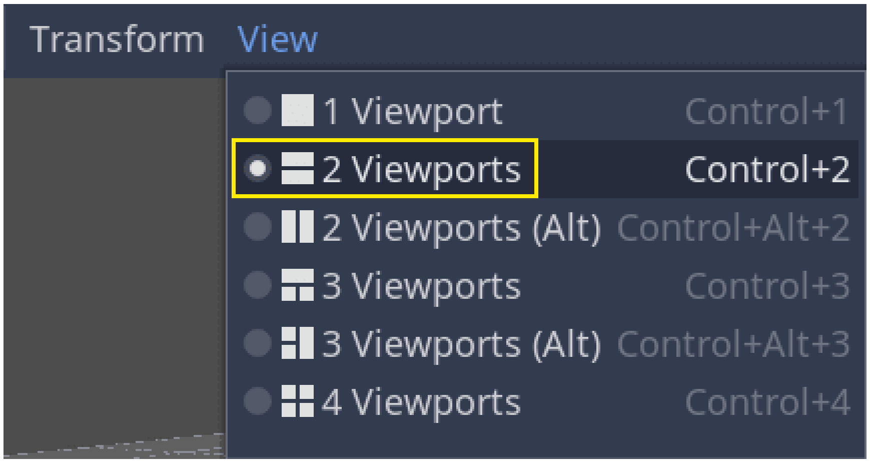
Notice the *Preview* checkbox that appears in the top-left of the 3D view when you have the *Camera* selected. You can click it to preview the in-game camera projection.


Perspective
Preview

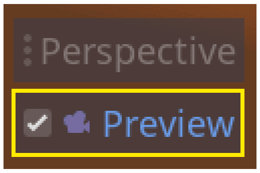
We're going to use the *Pivot* to rotate the camera as if it was on a crane. Let's first split the 3D view to be able to freely navigate the scene and see what the camera sees.

In the toolbar right above the viewport, click on *View*, then *2 Viewports*. You can also press `Ctrl + 2` (`Cmd + 2` on macOS).

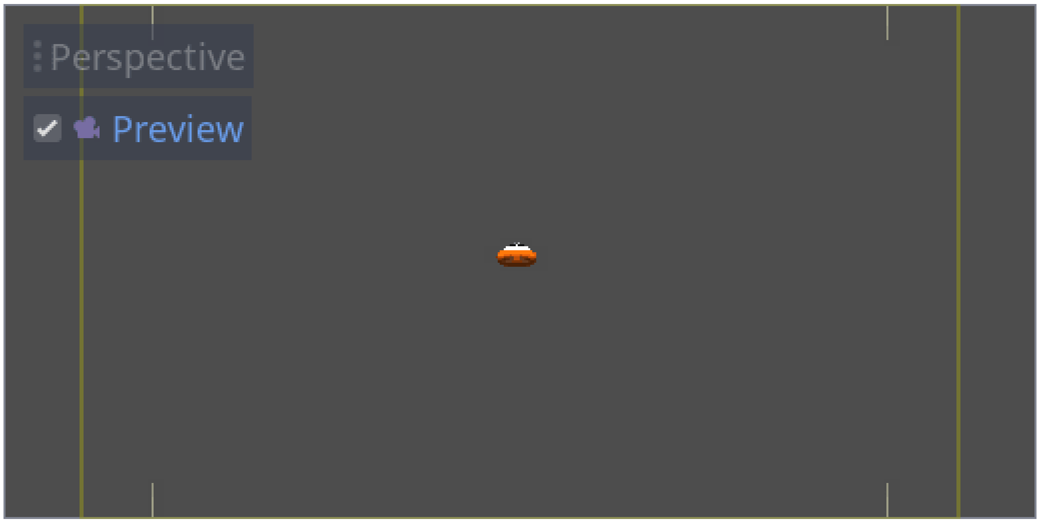

Scene
Import
Filter Node
Main
Ground
CollisionShape3d
MeshInstance3d
DirectionalLight3d
Player
CameraPivot
Camera3d
FileSystem
res://Player.tscn
Filter Files
main
Player
Transform
View
Top Orthogonal [auto]


Transform
View
1 Viewport Control+1
2 Viewports Control+2
2 Viewports (Alt) Control+Alt+2
3 Viewports Control+3
3 Viewports (Alt) Control+Alt+3
4 Viewports Control+4

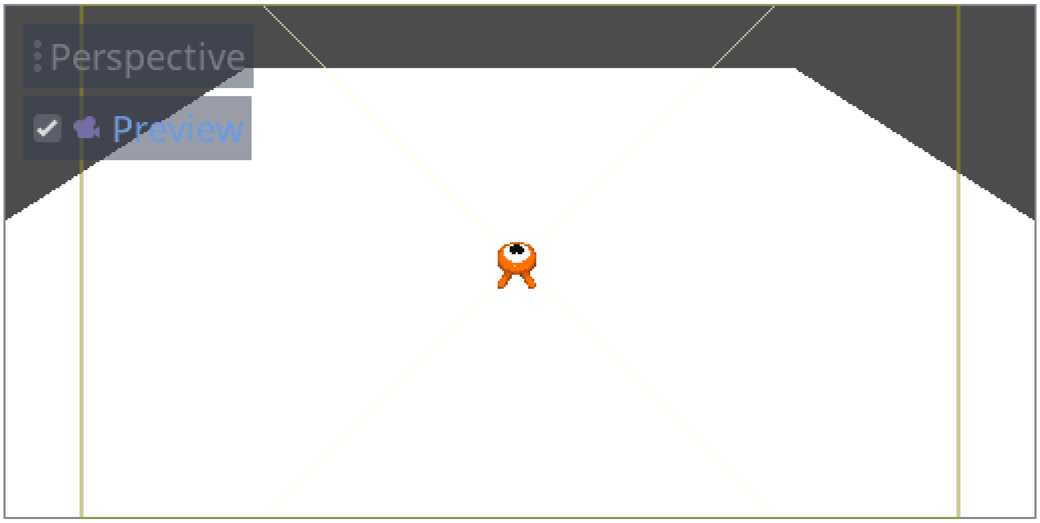
On the bottom view, select your [Camera3D]() and turn on camera Preview by clicking the checkbox.


Perspective
Preview

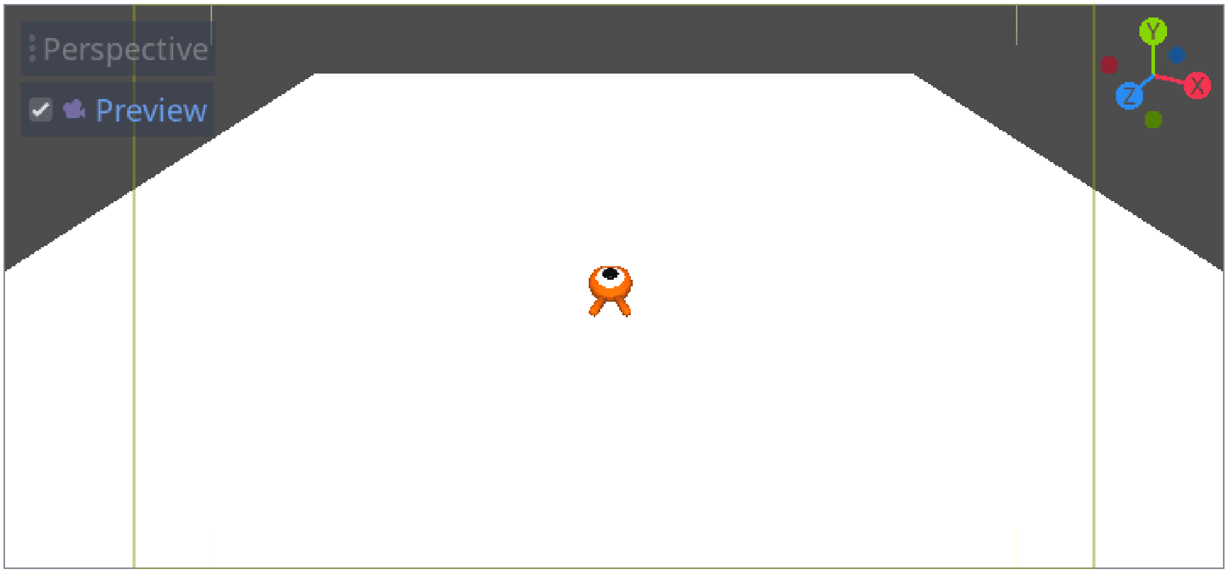
In the top view, make sure your *Camera3D* is selected and move the camera about `19` units on the Z axis (drag the blue arrow).


Perspective
☑ Preview

Here's where the magic happens. Select the *CameraPivot* and rotate it `-45` degrees around the X axis (using the red circle). You'll see the camera move as if it was attached to a crane.


Perspective
☑ Preview

You can run the scene by pressing `F6` and press the arrow keys to move the character.



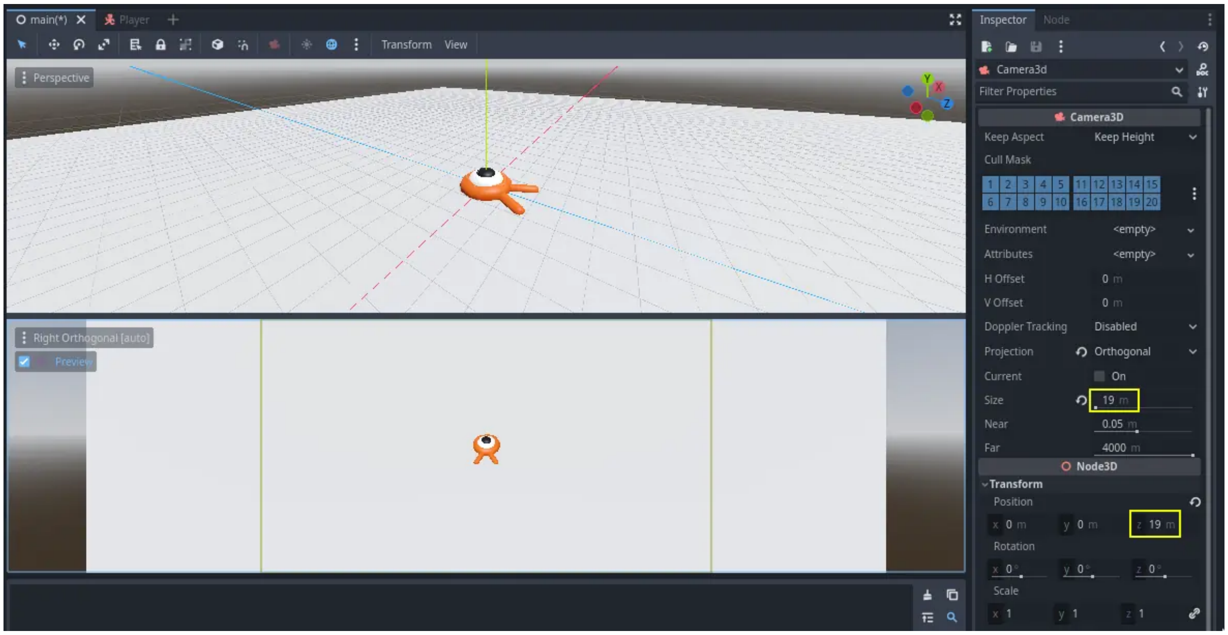
We can see some empty space around the character due to the perspective projection. In this game, we're going to use an orthographic projection instead to better frame the gameplay area and make it easier for the player to read distances.

Select the *Camera* again and in the *Inspector*, set the *Projection* to *Orthogonal* and the *Size* to `19`. The character should now look flatter and the ground should fill the background.

Note

When using an orthogonal camera in Godot 4, directional shadow quality is dependent on the camera's *Far* value. The higher the *Far* value, the further away the camera will be able to see. However, higher *Far* values also decrease shadow quality as the shadow rendering has to cover a greater distance.

If directional shadows look too blurry after switching to an orthogonal camera, decrease the camera's *Far* property to a lower value such as `100`. Don't decrease this *Far* property too much, or objects in the distance will start disappearing.


main(*)
Player
Transform
View
Perspective
Right Orthogonal [auto]
Preview
Inspector
Node
Camera3d
Filter Properties

Camera3D

Keep Aspect: Keep Height

Cull Mask: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Environment: <empty>

Attributes: <empty>

H Offset: 0 m

V Offset: 0 m

Doppler Tracking: Disabled

Projection: Orthogonal

Current: ☐ On

Size: 19 m

Near: 0.05 m

Far: 4000 m

Node3D

Transform

Position: x 0 m, y 0 m, z 19 m

Rotation: x 0°, y 0°, z 0°

Scale: x 1, y 1, z 1

Test your scene and you should be able to move in all 8 directions and not glitch through the floor!

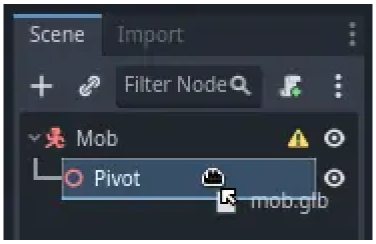
Ultimately, we have both player movement and the view in place. Next, we will work on the monsters.

Designing the mob scene

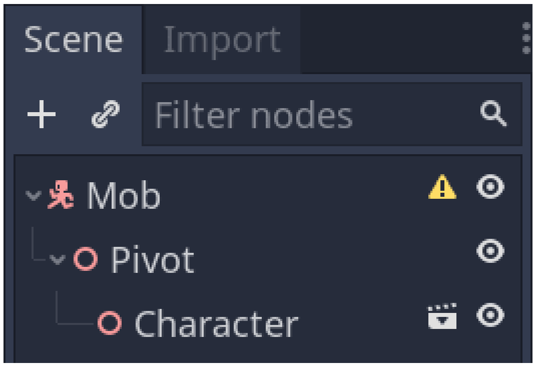
In this part, you're going to code the monsters, which we'll call mobs. In the next lesson, we'll spawn them randomly around the playable area.

Let's design the monsters themselves in a new scene. The node structure is going to be similar to the `player.tscn` scene.

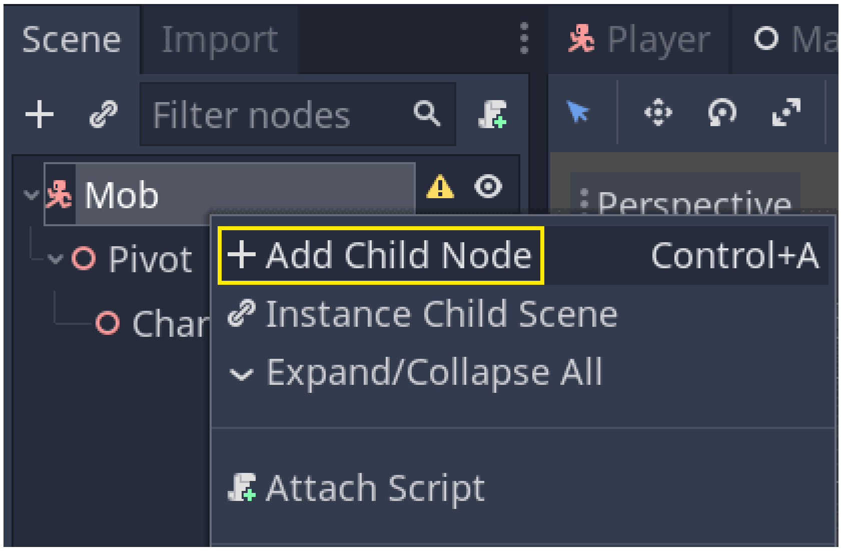
Create a scene with, once again, a CharacterBody3D node as its root. Name it `Mob`. Add a child node Node3D, name it `Pivot`. And drag and drop the file `mob.glb` from the *FileSystem* dock onto the `Pivot` to add the monster's 3D model to the scene.


Scene
Import
Filter Node
Mob
Pivot
mob.glb

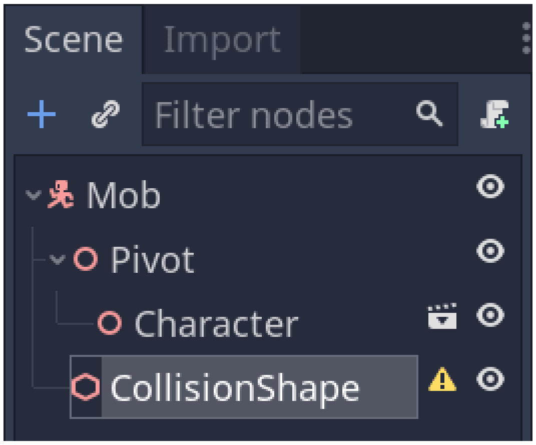
You can rename the newly created `mob` node into `Character`.


Scene
Import
Filter nodes
Mob
Pivot
Character

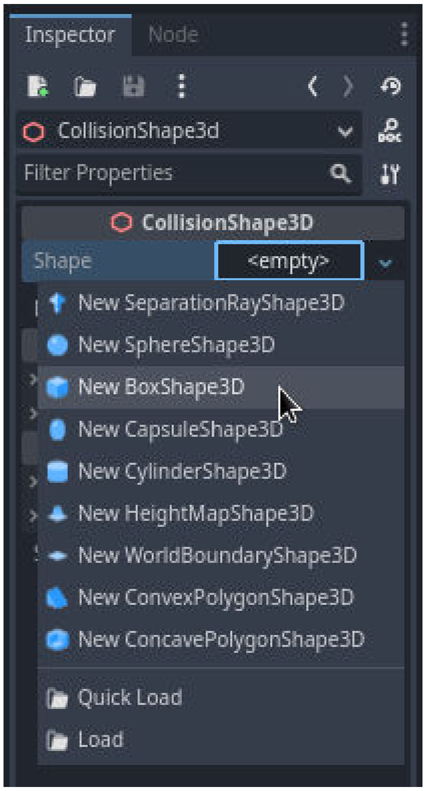
We need a collision shape for our body to work. Right-click on the `Mob` node, the scene's root, and click *Add Child Node*.


Scene
Import
Player
Ma
Filter nodes
Mob
Pivot
Char
Perspective
Add Child Node
Control+A
Instance Child Scene
Expand/Collapse All
Attach Script

Add a CollisionShape3D.

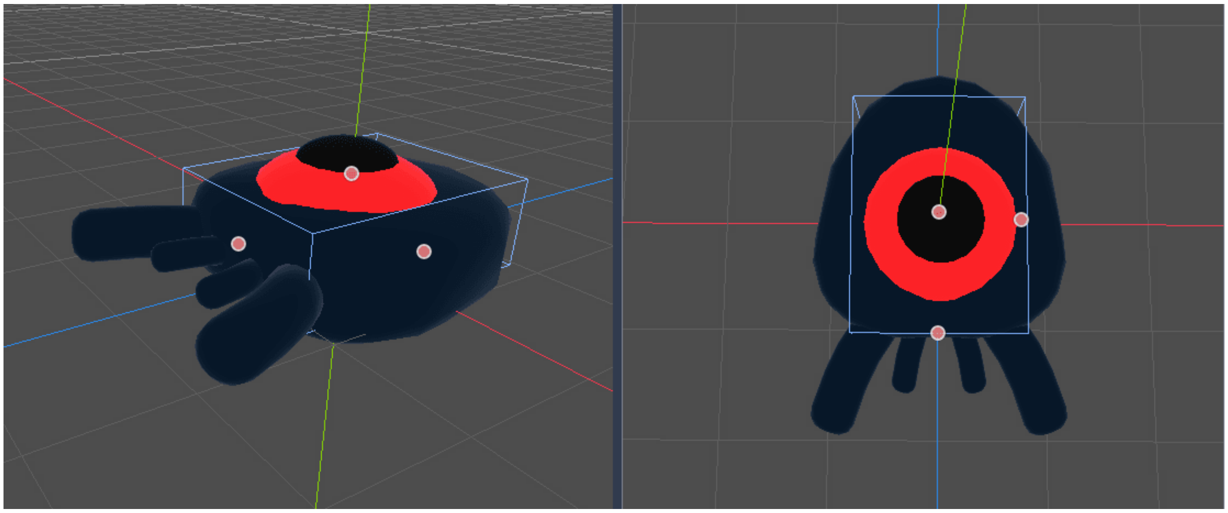

Scene
Import
Filter nodes
Mob
Pivot
Character
CollisionShape

In the *Inspector*, assign a *BoxShape3D* to the *Shape* property.


Inspector
Node
CollisionShape3d
Filter Properties
CollisionShape3D
Shape
<empty>
New SeparationRayShape3D
New SphereShape3D
New BoxShape3D
New CapsuleShape3D
New CylinderShape3D
New HeightMapShape3D
New WorldBoundaryShape3D
New ConvexPolygonShape3D
New ConcavePolygonShape3D
Quick Load
Load

We should change its size to fit the 3D model better. You can do so interactively by clicking and dragging on the orange dots.

The box should touch the floor and be a little thinner than the model. Physics engines work in such a way that if the player's sphere touches even the box's corner, a collision will occur. If the box is a little too big compared to the 3D model, you may die at a distance from the monster, and the game will feel unfair to the players.



Notice that my box is taller than the monster. It is okay in this game because we're looking at the scene from above and using a fixed perspective. Collision shapes don't have to match the model exactly. It's the way the game feels when you test it that should dictate their form and size.

Removing monsters off-screen

We're going to spawn monsters at regular time intervals in the game level. If we're not careful, their count could increase to infinity, and we don't want that. Each mob instance has both a memory and a processing cost, and we don't want to pay for it when the mob is outside the screen.

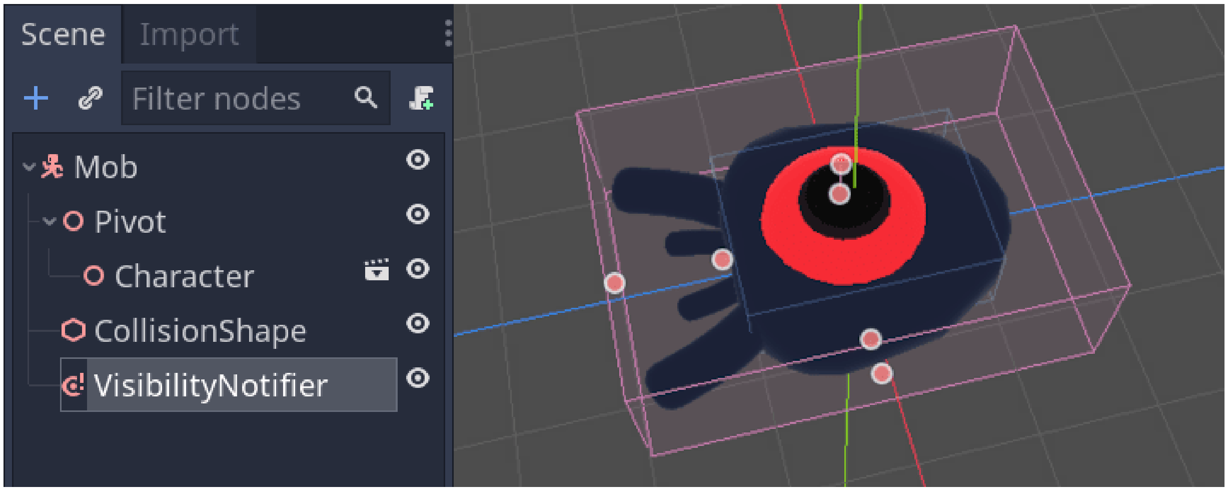
Once a monster leaves the screen, we don't need it anymore, so we should delete it. Godot has a node that detects when objects leave the screen, VisibleOnScreenNotifier3D, and we're going to use it to destroy our mobs.

Note

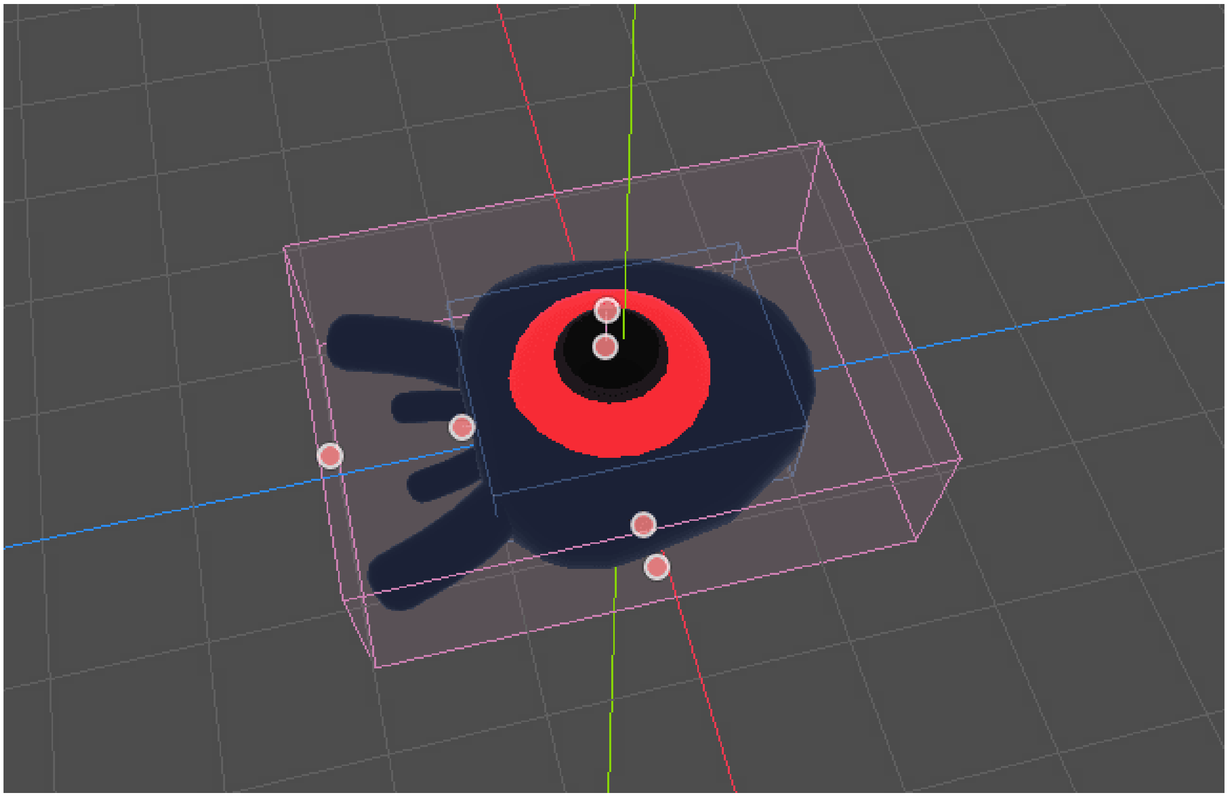
When you keep instancing an object, there's a technique you can use to avoid the cost of creating and destroying instances all the time called pooling. It consists of pre-creating an array of objects and reusing them over and over.

When working with GDScript, you don't need to worry about this. The main reason to use pools is to avoid freezes with garbage-collected languages like C# or Lua. GDScript uses a different technique to manage memory, reference counting, which doesn't have that caveat. You can learn more about that here: Memory management.

Select the `Mob` node and add a child node VisibleOnScreenNotifier3D. Another box, pink this time, appears. When this box completely leaves the screen, the node will emit a signal.


Scene
Import
Filter nodes
Mob
Pivot
Character
CollisionShape
VisibilityNotifier

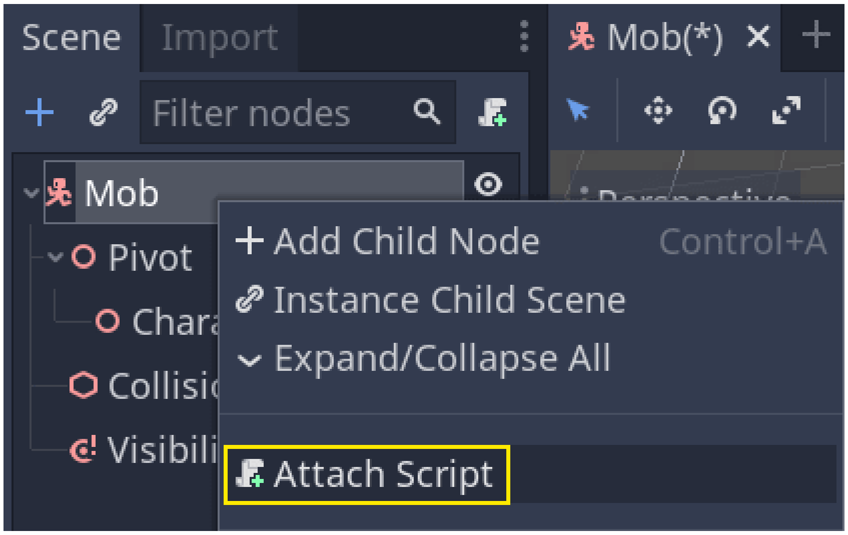
Resize it using the orange dots until it covers the entire 3D model.



Coding the mob's movement

Let's implement the monster's motion. We're going to do this in two steps. First, we'll write a script on the `Mob` that defines a function to initialize the monster. We'll then code the randomized spawn mechanism in the `main.tscn` scene and call the function from there.

Attach a script to the `Mob`.


Scene
Import
Mob(*)
Filter nodes
Mob
Pivot
Chara
Collisi
Visibili
Perspective
Add Child Node
Control+A
Instance Child Scene
Expand/Collapse All
Attach Script

Here's the movement code to start with. We define two properties, `min_speed` and `max_speed`, to define a random speed range, which we will later use to define `CharacterBody3D.velocity`.

GDScript

```
extends CharacterBody3D

# Minimum speed of the mob in meters per second.
@export var min_speed = 10
# Maximum speed of the mob in meters per second.
@export var max_speed = 18


func _physics_process(_delta):
    move_and_slide()
```

C#

```
using Godot;

public partial class Mob : CharacterBody3D
{
    // Don't forget to rebuild the project so the editor knows
about the new export variable.

    // Minimum speed of the mob in meters per second
    [Export]
    public int MinSpeed { get; set; } = 10;
    // Maximum speed of the mob in meters per second
    [Export]
    public int MaxSpeed { get; set; } = 18;

    public override void _PhysicsProcess(double delta)
    {
        MoveAndSlide();
    }
}
```

Similarly to the player, we move the mob every frame by calling the function `CharacterBody3D.move_and_slide()`. This time, we don't update the `velocity` every frame; we want the monster to move at a constant speed and leave the screen, even if it were to hit an obstacle.

We need to define another function to calculate the `CharacterBody3D.velocity`. This function will turn the monster towards the player and randomize both its angle of motion and its velocity.

The function will take a `start_position`,the mob's spawn position, and the `player_position` as its arguments.

We position the mob at `start_position` and turn it towards the player using the `look_at_from_position()` method, and randomize the angle by rotating a random amount around the Y axis. Below, `randf_range()` outputs a random value between `-PI / 4` radians and `PI / 4` radians.

GDScript

```
# This function will be called from the Main scene.
func initialize(start_position, player_position):
    # We position the mob by placing it at start_position
    # and rotate it towards player_position, so it looks at the
```

```
player.
    look_at_from_position(start_position, player_position,
Vector3.UP)
    # Rotate this mob randomly within range of -45 and +45
degrees,
    # so that it doesn't move directly towards the player.
    rotate_y(randf_range(-PI / 4, PI / 4))
```

C#

```
// This function will be called from the Main scene.
public void Initialize(Vector3 startPosition, Vector3
playerPosition)
{
    // We position the mob by placing it at startPosition
    // and rotate it towards playerPosition, so it looks at the
player.
    LookAtFromPosition(startPosition, playerPosition,
Vector3.Up);
    // Rotate this mob randomly within range of -45 and +45
degrees,
    // so that it doesn't move directly towards the player.
    RotateY((float)GD.RandRange(-Mathf.Pi / 4.0, Mathf.Pi /
4.0));
}
```

We got a random position, now we need a `random_speed`. `randi_range()` will be useful as it gives random int values, and we will use `min_speed` and `max_speed`. `random_speed` is just an integer, and we just use it to multiply our `CharacterBody3D.velocity`. After `random_speed` is applied, we rotate `CharacterBody3D.velocity` Vector3 towards the player.

GDScript

```
func initialize(start_position, player_position):
    # ...

    # We calculate a random speed (integer)
    var random_speed = randi_range(min_speed, max_speed)
    # We calculate a forward velocity that represents the
speed.
    velocity = Vector3.FORWARD * random_speed
    # We then rotate the velocity vector based on the mob's Y
rotation
    # in order to move in the direction the mob is looking.
    velocity = velocity.rotated(Vector3.UP, rotation.y)
```

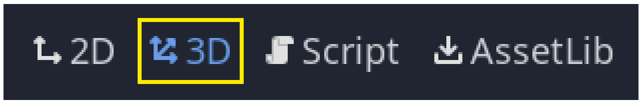
C#

```
public void Initialize(Vector3 startPosition, Vector3
playerPosition)
{
    // ...

    // We calculate a random speed (integer).
    int randomSpeed = GD.RandRange(MinSpeed, MaxSpeed);
    // We calculate a forward velocity that represents the
speed.
    Velocity = Vector3.Forward * randomSpeed;
    // We then rotate the velocity vector based on the mob's Y
rotation
    // in order to move in the direction the mob is looking.
    Velocity = Velocity.Rotated(Vector3.Up, Rotation.Y);
}
```

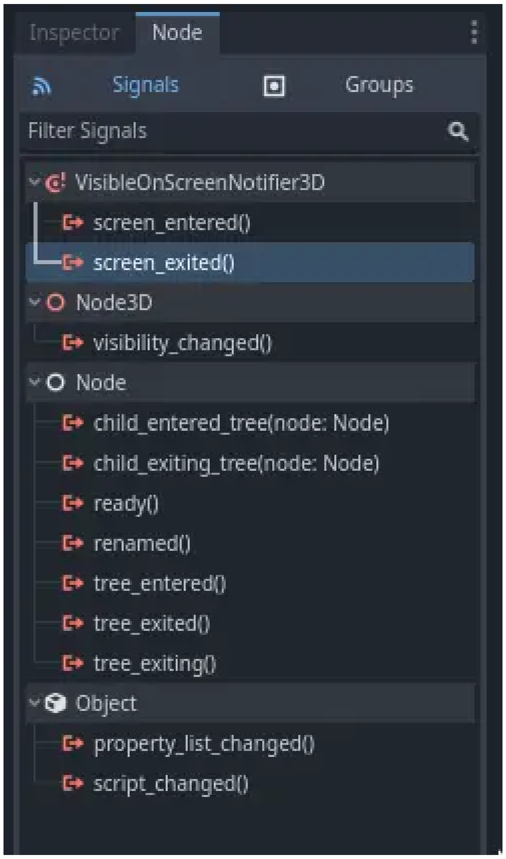
Leaving the screen

We still have to destroy the mobs when they leave the screen. To do so, we'll connect our VisibleOnScreenNotifier3D node's `screen_exited` signal to the `Mob`.

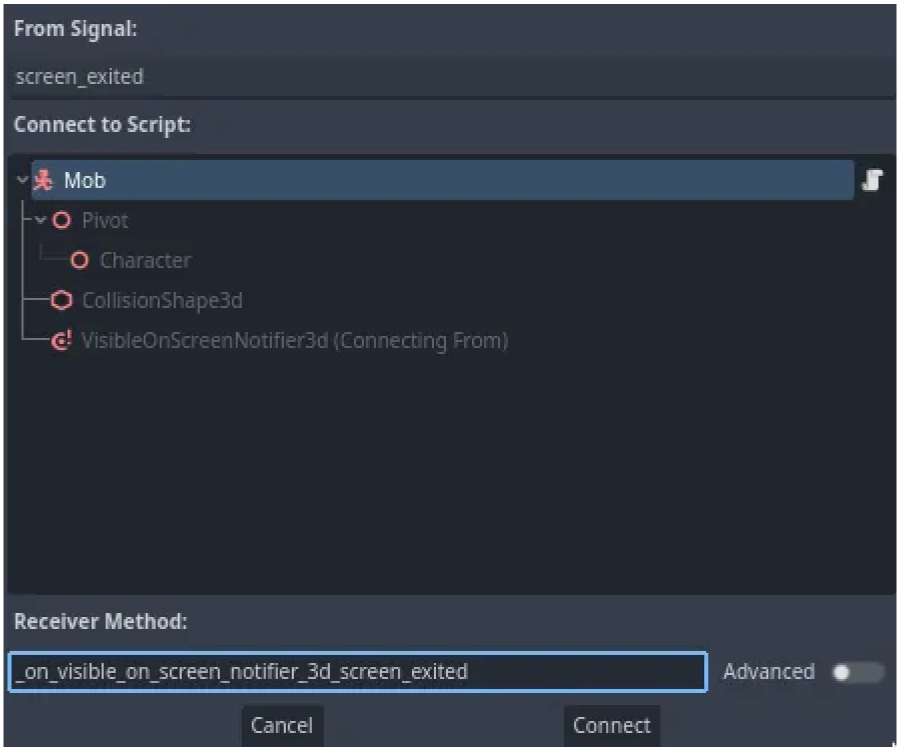
Head back to the 3D viewport by clicking on the *3D* label at the top of the editor. You can also press `Ctrl + F2` (`Opt + 2` on macOS).



Select the VisibleOnScreenNotifier3D node and on the right side of the interface, navigate to the *Node* dock. Double-click the `screen_exited()` signal.


Inspector
Node
Signals
Groups
Filter Signals
VisibleOnScreenNotifier3D
screen_entered()
screen_exited()
Node3D
visibility_changed()
Node
child_entered_tree(node: Node)
child_exiting_tree(node: Node)
ready()
renamed()
tree_entered()
tree_exited()
tree_exiting()
Object
property_list_changed()
script_changed()

Connect the signal to the `Mob`



From Signal: screen_exited

Receiver Method: _on_visible_on_screen_notifier_3d_screen_exited

Connect to Script:
Mob
Pivot
Character
CollisionShape3d
VisibleOnScreenNotifier3d (Connecting From)
Advanced
Cancel
Connect

This will add a new function for you in your mob script, `_on_visible_on_screen_notifier_3d_screen_exited()`. From it, call the `queue_free()` method. This function destroys the instance it's called on.

GDScript

```
func _on_visible_on_screen_notifier_3d_screen_exited():
    queue_free()
```

C#

```
// We also specified this function name in PascalCase in the
editor's connection window.
private void OnVisibilityNotifierScreenExited()
{
    QueueFree();
}
```

Our monster is ready to enter the game! In the next part, you will spawn monsters in the game level.

Here is the complete `mob.gd` script for reference.

GDScript

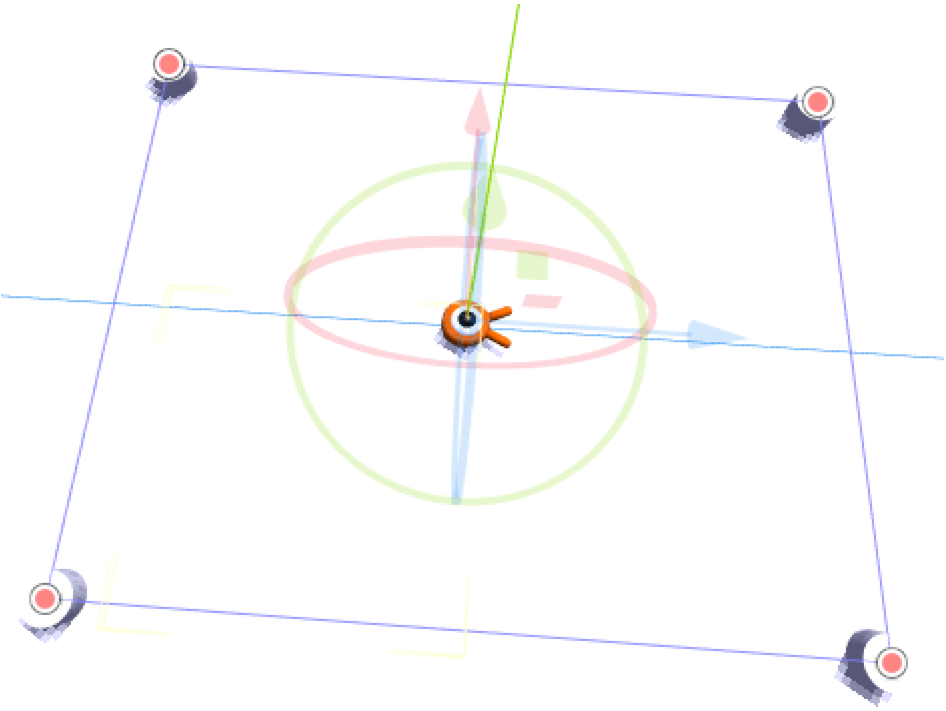
```
extends CharacterBody3D

# Minimum speed of the mob in meters per second.
@export var min_speed = 10
# Maximum speed of the mob in meters per second.
@export var max_speed = 18

func _physics_process(_delta):
    move_and_slide()

# This function will be called from the Main scene.
func initialize(start_position, player_position):
    # We position the mob by placing it at start_position
    # and rotate it towards player_position, so it looks at the player.
    look_at_from_position(start_position, player_position, Vector3.UP)
    # Rotate this mob randomly within range of -45 and +45 degrees,
    # so that it doesn't move directly towards the player.
    rotate_y(randf_range(-PI / 4, PI / 4))

    # We calculate a random speed (integer)
    var random_speed = randi_range(min_speed, max_speed)
    # We calculate a forward velocity that represents the speed.
    velocity = Vector3.FORWARD * random_speed
    # We then rotate the velocity vector based on the mob's Y rotation
    # in order to move in the direction the mob is looking.
    velocity = velocity.rotated(Vector3.UP, rotation.y)

func _on_visible_on_screen_notifier_3d_screen_exited():
    queue_free()
```

C#

```
using Godot;

public partial class Mob : CharacterBody3D
{
    // Minimum speed of the mob in meters per second.
    [Export]
    public int MinSpeed { get; set; } = 10;
    // Maximum speed of the mob in meters per second.
    [Export]
    public int MaxSpeed { get; set; } = 18;

    public override void _PhysicsProcess(double delta)
    {
        MoveAndSlide();
    }

    // This function will be called from the Main scene.
    public void Initialize(Vector3 startPosition, Vector3
playerPosition)
    {
        // We position the mob by placing it at startPosition
        // and rotate it towards playerPosition, so it looks at
the player.
        LookAtFromPosition(startPosition, playerPosition,
Vector3.Up);
        // Rotate this mob randomly within range of -45 and +45
degrees,
        // so that it doesn't move directly towards the player.
        RotateY((float)GD.RandRange(-Mathf.Pi / 4.0, Mathf.Pi /
4.0));

        // We calculate a random speed (integer).
        int randomSpeed = GD.RandRange(MinSpeed, MaxSpeed);
        // We calculate a forward velocity that represents the
speed.
        Velocity = Vector3.Forward * randomSpeed;
        // We then rotate the velocity vector based on the
mob's Y rotation
        // in order to move in the direction the mob is
looking.
        Velocity = Velocity.Rotated(Vector3.Up, Rotation.Y);
    }
    // We also specified this function name in PascalCase in
the editor's connection window.
    private void OnVisibilityNotifierScreenExited()
    {
        QueueFree();
    }
}
```

Spawning monsters

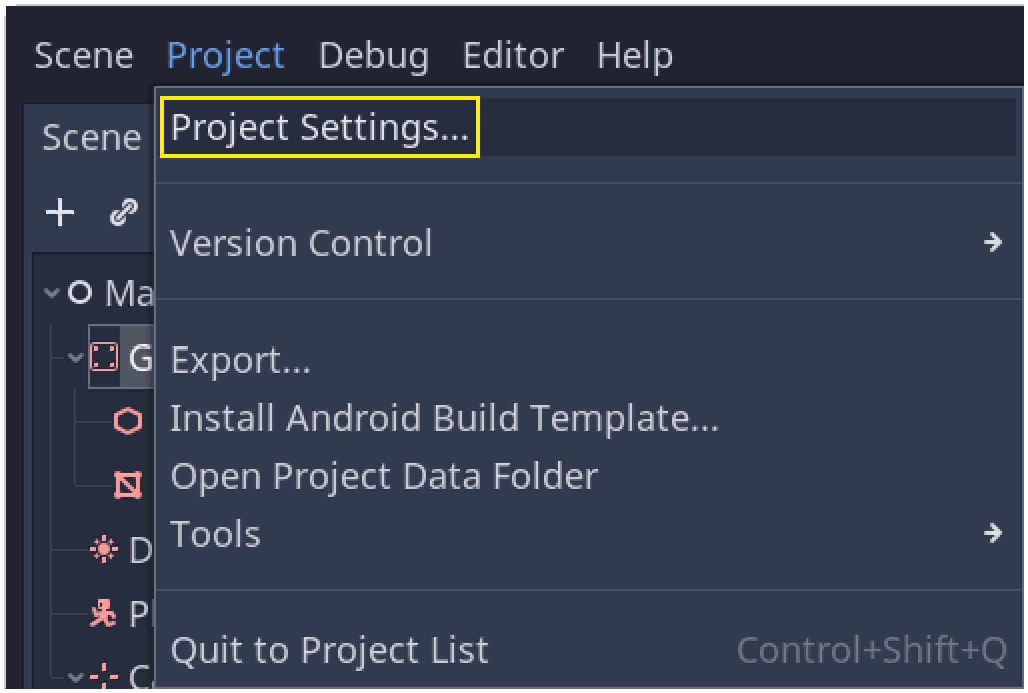
In this part, we're going to spawn monsters along a path randomly. By the end, you will have monsters roaming the game board.



Double-click on `main.tscn` in the *FileSystem* dock to open the `Main` scene.

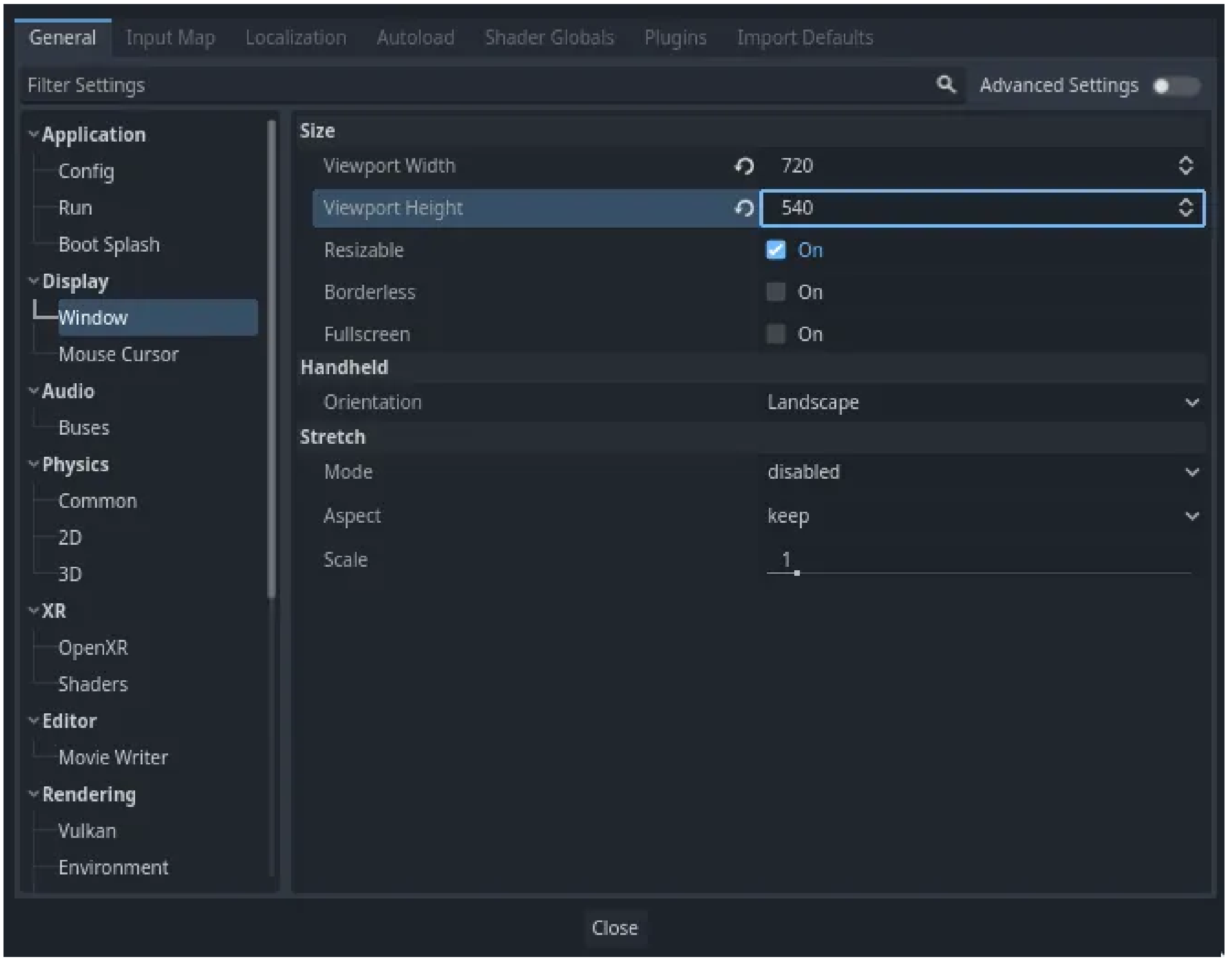
Before drawing the path, we're going to change the game resolution. Our game has a default window size of `1152x648`. We're going to set it to `720x540`, a nice little box.

Go to *Project -> Project Settings*.


Scene
Project
Debug
Editor
Help
Scene
Ma
G
D
Pl
C
Project Settings...
Version Control
Export...
Install Android Build Template...
Open Project Data Folder
Tools
Quit to Project List
Control+Shift+Q

If you still have *Input Map* open, switch to the *General* tab.

In the left menu, navigate down to *Display -> Window*. On the right, set the *Width* to `720` and the *Height* to `540`.


General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Filter Settings
Advanced Settings
Application
Config
Run
Boot Splash
Display
Window
Mouse Cursor
Audio
Buses
Physics
Common
2D
3D
XR
OpenXR
Shaders
Editor
Movie Writer
Rendering
Vulkan
Environment
Close

Size

Viewport Width: 720

Viewport Height: 540

Resizable: ☑ On

Borderless: ☐ On

Fullscreen: ☐ On

Handheld

Orientation: Landscape

Stretch

Mode: disabled

Aspect: keep

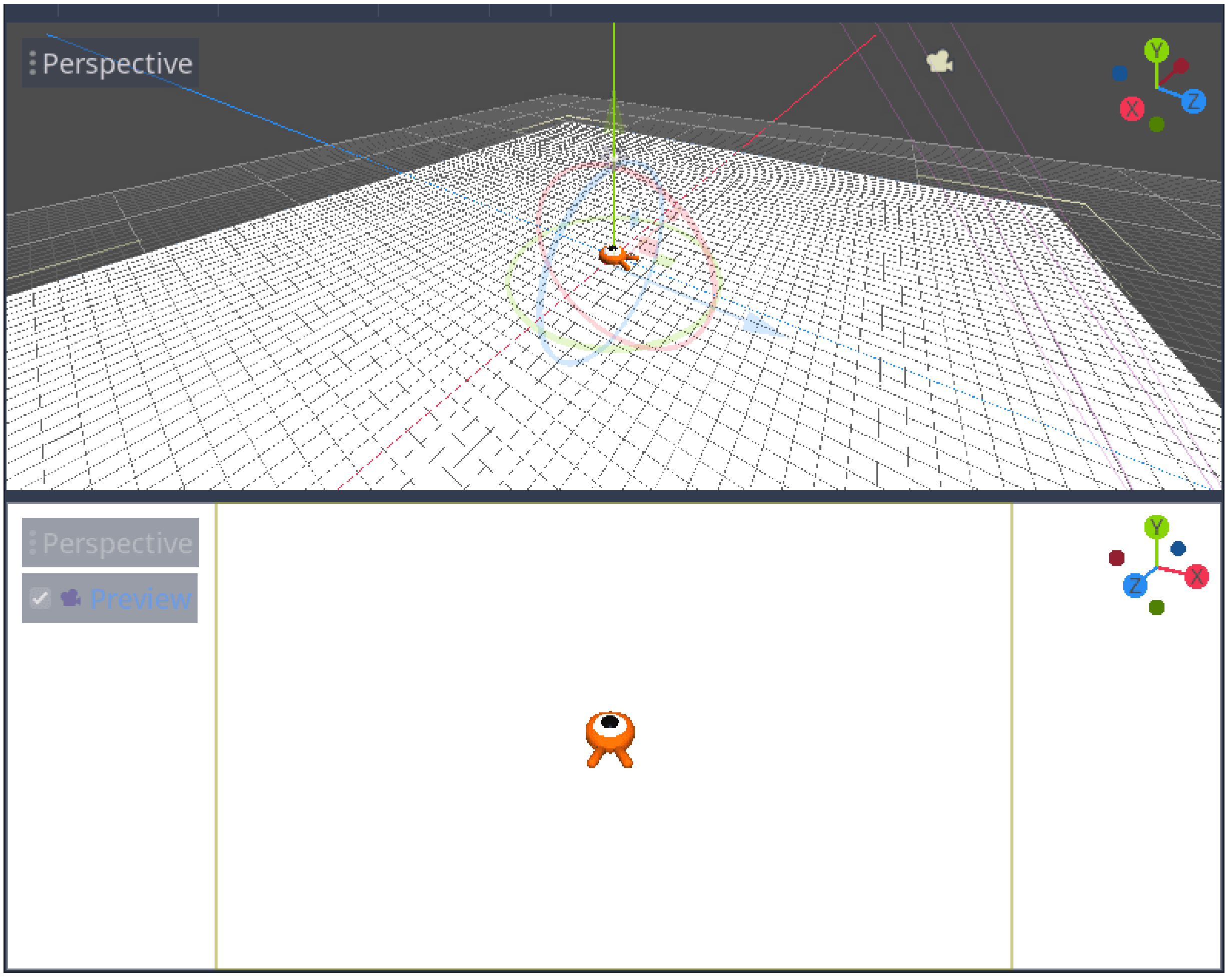
Scale: 1

Creating the spawn path

Like you did in the 2D game tutorial, you're going to design a path and use a PathFollow3D node to sample random locations on it.

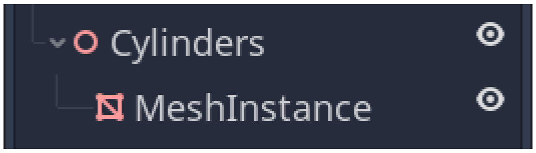
In 3D though, it's a bit more complicated to draw the path. We want it to be around the game view so monsters appear right outside the screen. But if we draw a path, we won't see it from the camera preview.

To find the view's limits, we can use some placeholder meshes. Your viewport should still be split into two parts, with the camera preview at the bottom. If that isn't the case, press `Ctrl + 2` (`Cmd + 2` on macOS) to split the view into two. Select the Camera3D node and click the *Preview* checkbox in the bottom viewport.

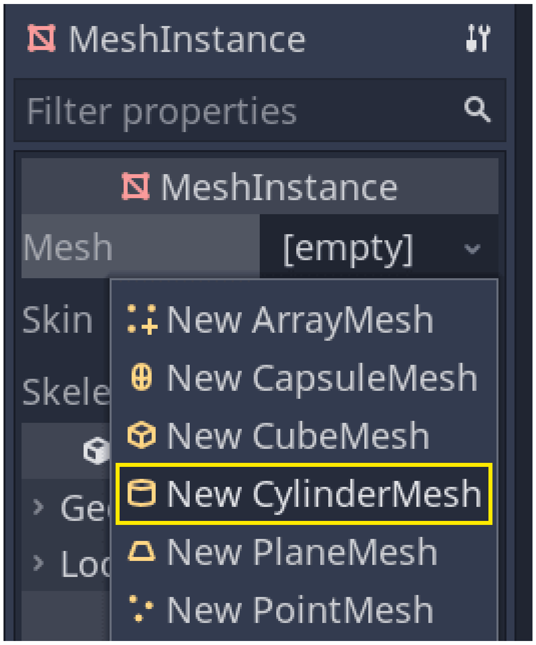

Perspective
Y
X
Z
Perspective
☑ Preview
Y
Z
X

Adding placeholder cylinders

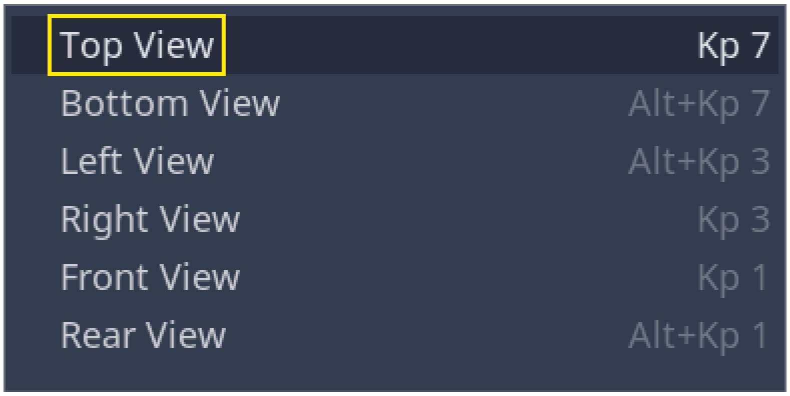
Let's add the placeholder meshes. Add a new Node3D as a child of the `Main` node and name it `Cylinders`. We'll use it to group the cylinders. Select `Cylinders` and add a child node MeshInstance3D


Cylinders
MeshInstance

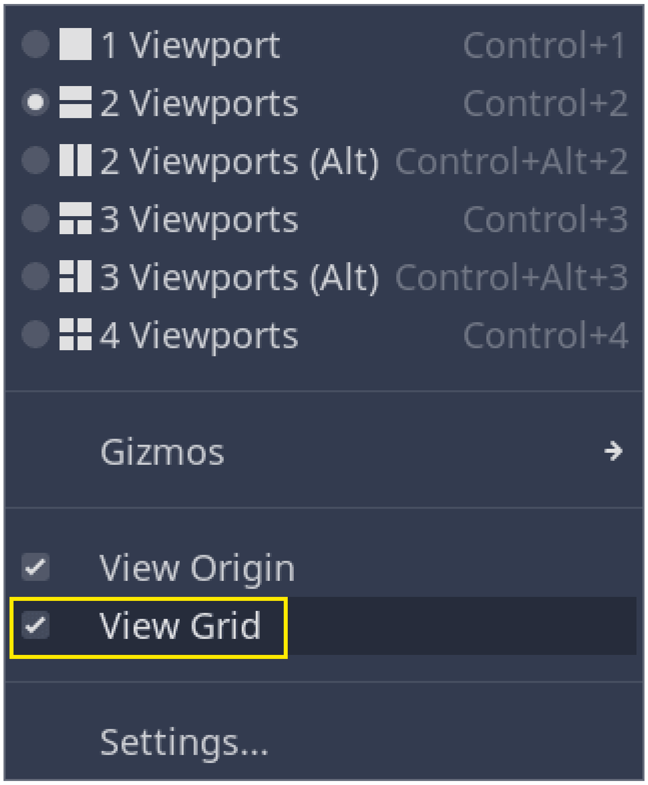
In the *Inspector*, assign a *CylinderMesh* to the *Mesh* property.


MeshInstance
Filter properties
MeshInstance
Mesh
[empty]
Skin
Skele
Ge
Loc
New ArrayMesh
New CapsuleMesh
New CubeMesh
New CylinderMesh
New PlaneMesh
New PointMesh

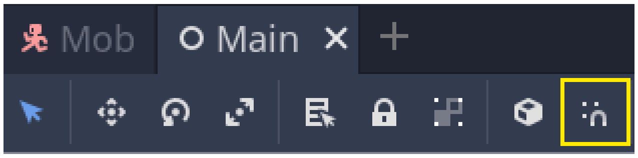
Set the top viewport to the top orthogonal view using the menu in the viewport's top-left corner. Alternatively, you can press the keypad's 7 key.


Top View Kp 7
Bottom View Alt+Kp 7
Left View Alt+Kp 3
Right View Kp 3
Front View Kp 1
Rear View Alt+Kp 1

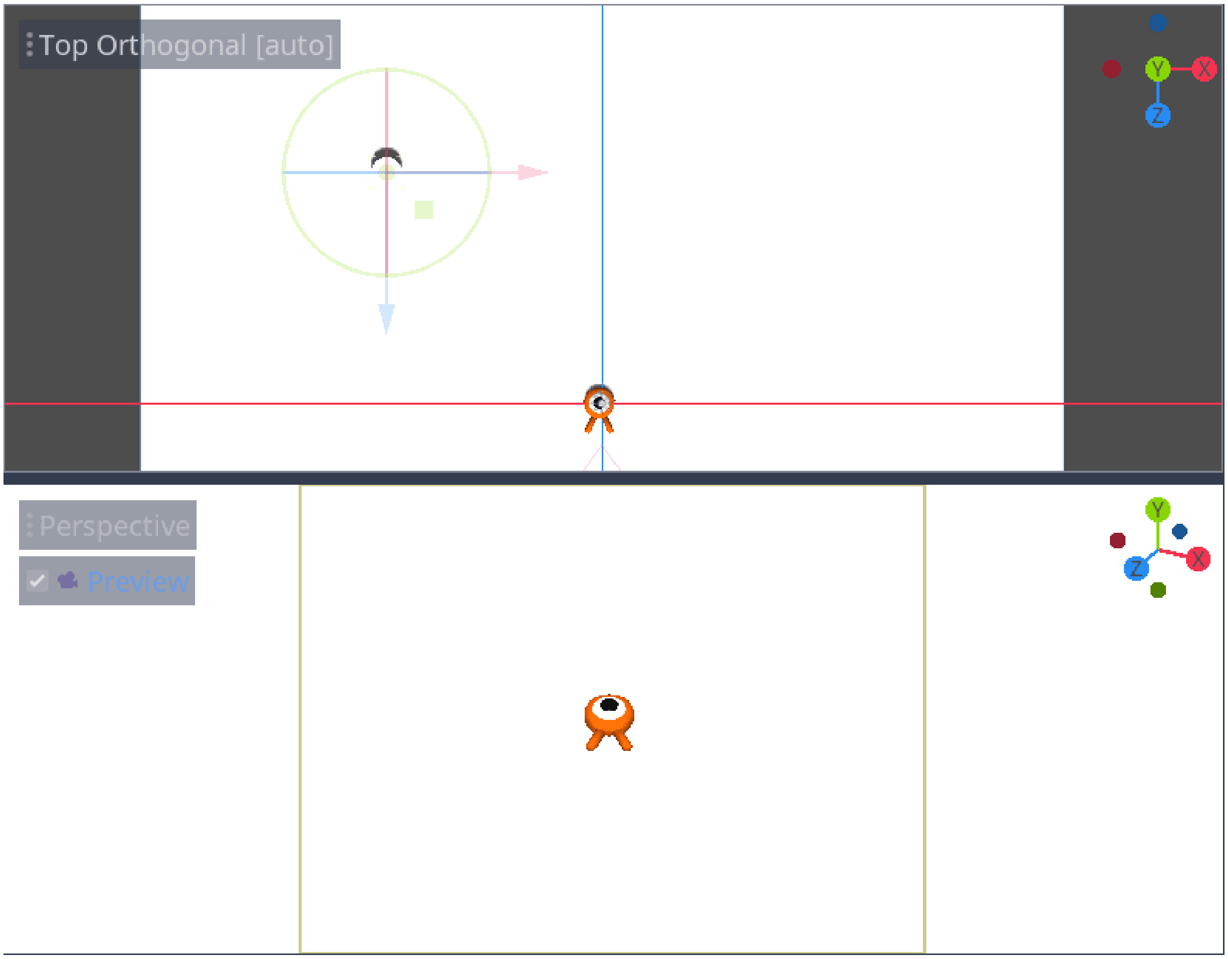
The grid may be distracting. You can toggle it by going to the *View* menu in the toolbar and clicking *View Grid*.


1 Viewport Control+1
2 Viewports Control+2
2 Viewports (Alt) Control+Alt+2
3 Viewports Control+3
3 Viewports (Alt) Control+Alt+3
4 Viewports Control+4
Gizmos
View Origin
View Grid
Settings...

You now want to move the cylinder along the ground plane, looking at the camera preview in the bottom viewport. I recommend using grid snap to do so. You can toggle it by clicking the magnet icon in the toolbar or pressing Y.

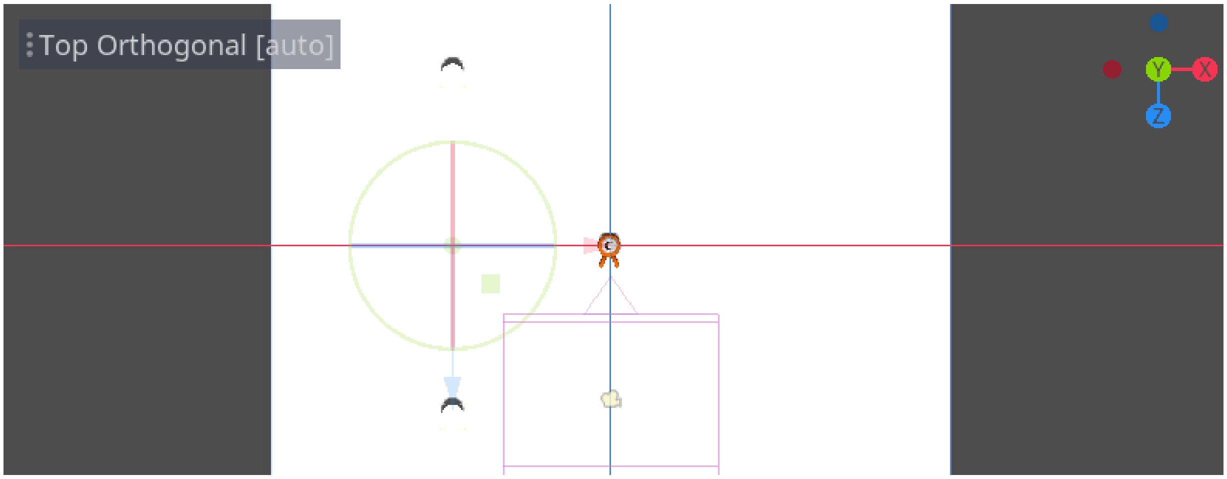


Move the cylinder so it's right outside the camera's view in the top-left corner.

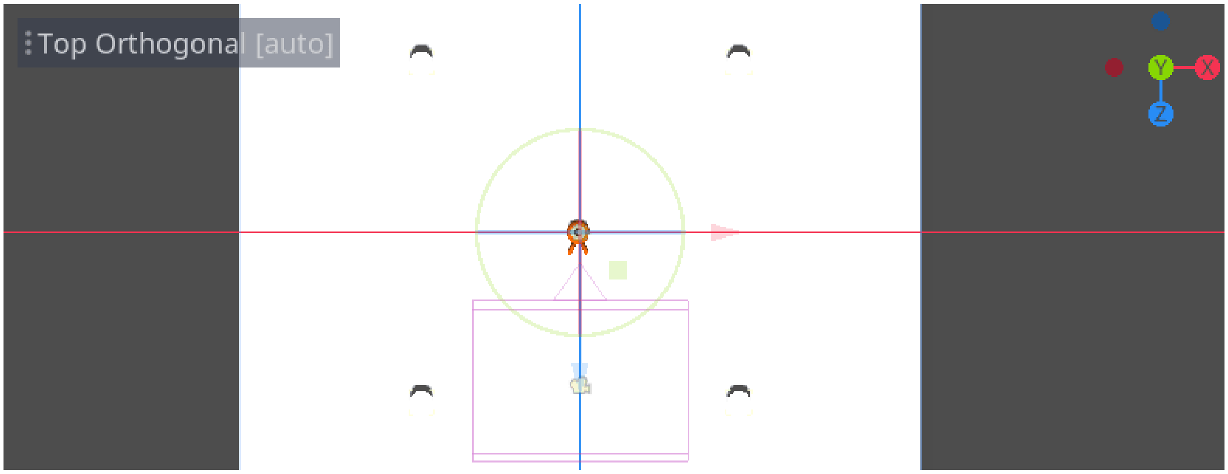

Top Orthogonal [auto]
Y
X
Z
Perspective
Preview

We're going to create copies of the mesh and place them around the game area. Press `Ctrl + D` (`Cmd + D` on macOS) to duplicate the node. You can also right-click the node in the *Scene* dock and select *Duplicate*. Move the copy down along the blue Z axis until it's right outside the camera's preview.

Select both cylinders by pressing the `Shift` key and clicking on the unselected one and duplicate them.



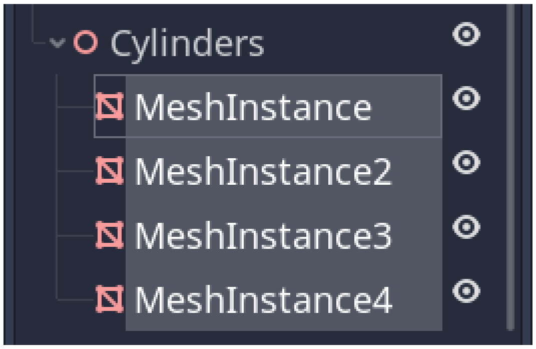
Move them to the right by dragging the red X axis.



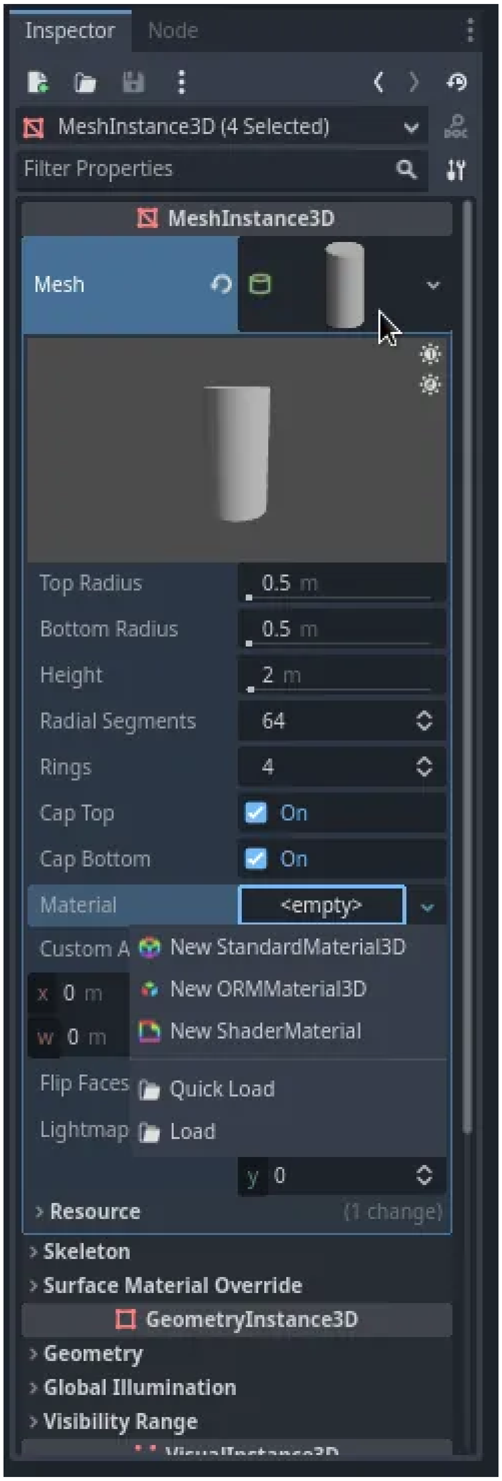
They're a bit hard to see in white, aren't they? Let's make them stand out by giving them a new material.

In 3D, materials define a surface's visual properties like its color, how it reflects light, and more. We can use them to change the color of a mesh.

We can update all four cylinders at once. Select all the mesh instances in the *Scene* dock. To do so, you can click on the first one and Shift click on the last one.


Cylinders
MeshInstance
MeshInstance2
MeshInstance3
MeshInstance4

In the *Inspector*, expand the *Material* section and assign a StandardMaterial3D to slot *0*.


Inspector
Node
MeshInstance3D (4 Selected)
Filter Properties
MeshInstance3D
New StandardMaterial3D
New ORMMaterial3D
New ShaderMaterial
Quick Load
Load
Resource
(1 change)
Skeleton
Surface Material Override
GeometryInstance3D
Geometry
Global Illumination
Visibility Range

Mesh

Top Radius 0.5 m

Bottom Radius 0.5 m

Height 2 m

Radial Segments 64

Rings 4

Cap Top ☑ On

Cap Bottom ☑ On

Material <empty>

Custom A

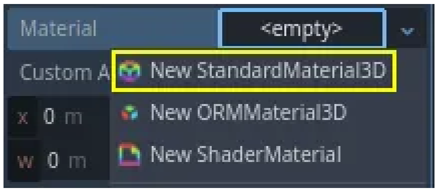
x 0 m

w 0 m

Flip Faces

Lightmap

y 0


New StandardMaterial3D
New ORMMaterial3D
New ShaderMaterial

Material <empty>

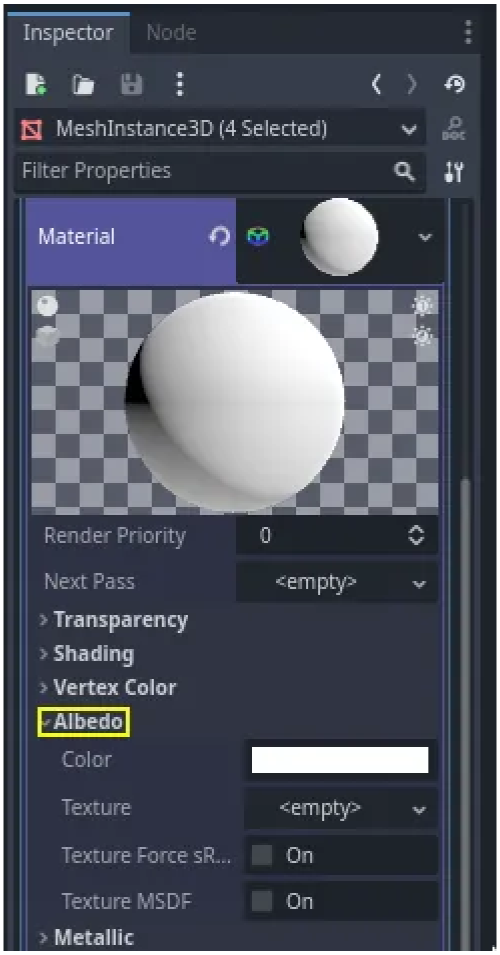
Custom A

x 0 m

w 0 m

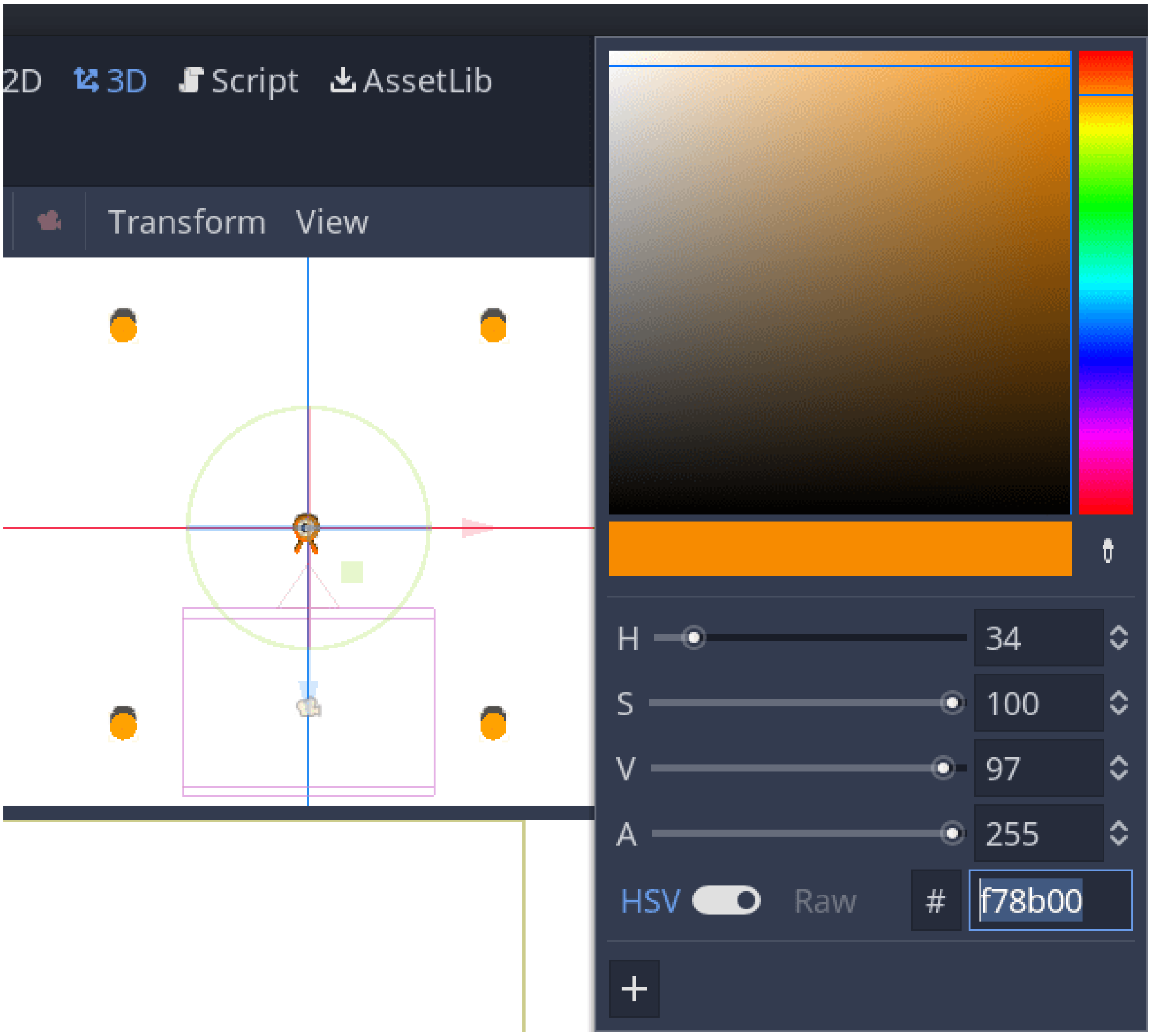
Click the sphere icon to open the material resource. You get a preview of the material and a long list of sections filled with properties. You can use these to create all sorts of surfaces, from metal to rock or water.

Expand the *Albedo* section.


Inspector
Node
MeshInstance3D (4 Selected)
Filter Properties
Material
Transparency
Shading
Vertex Color
Albedo
Metallic

Render Priority 0
Next Pass <empty>
Color
Texture <empty>
Texture Force sR... ☐ On
Texture MSDF ☐ On

Set the color to something that contrasts with the background, like a bright orange.


2D
3D
Script
AssetLib
Transform
View
+

H 34

S 100

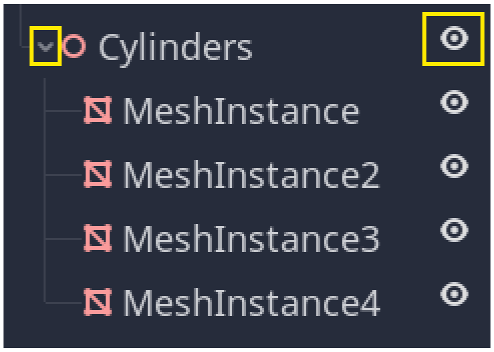
V 97

A 255

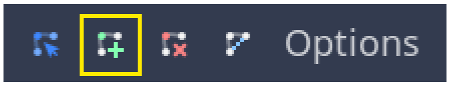
HSV Raw

f78b00

We can now use the cylinders as guides. Fold them in the *Scene* dock by clicking the grey arrow next to them. Moving forward, you can also toggle their visibility by clicking the eye icon next to *Cylinders*.


Cylinders
MeshInstance
MeshInstance2
MeshInstance3
MeshInstance4

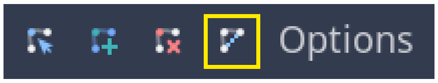
Add a child node Path3D to `Main` node. In the toolbar, four icons appear. Click the *Add Point* tool, the icon with the green "+" sign.



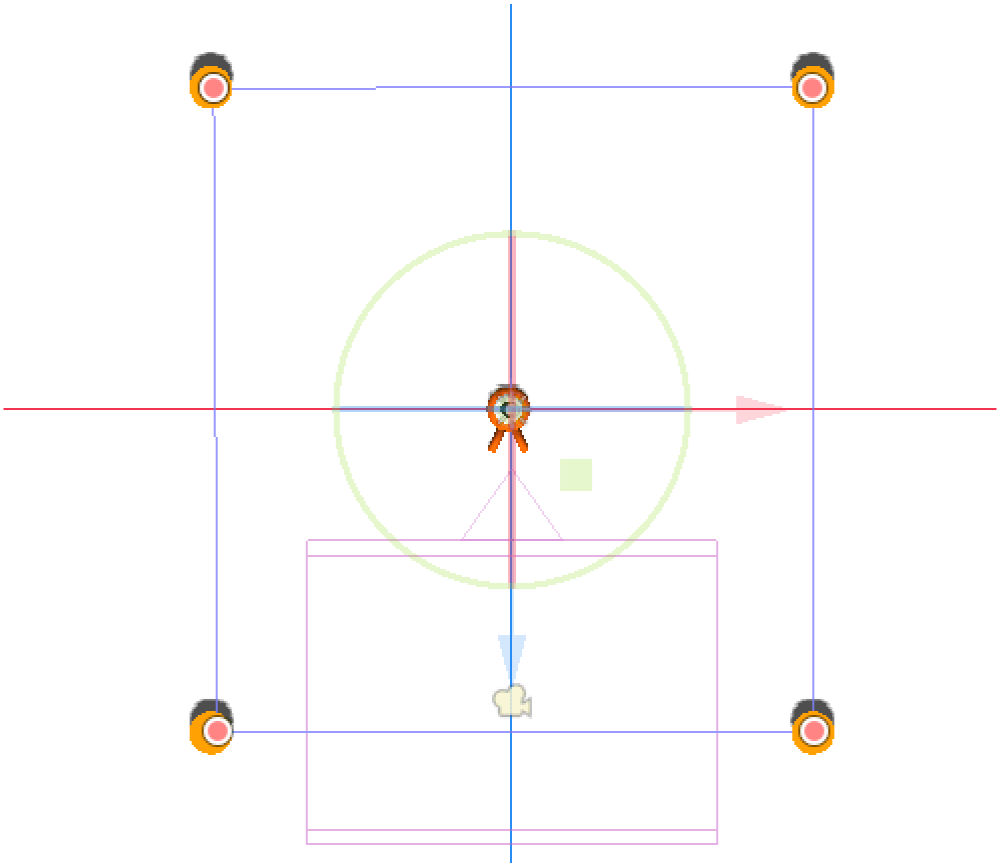
Note

You can hover any icon to see a tooltip describing the tool.

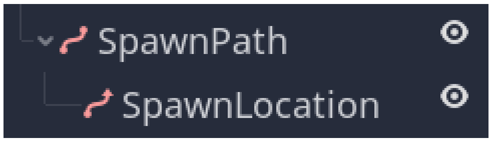
Click in the center of each cylinder to create a point. Then, click the *Close Curve* icon in the toolbar to close the path. If any point is a bit off, you can click and drag on it to reposition it.



Your path should look like this.



To sample random positions on it, we need a PathFollow3D node. Add a PathFollow3D as a child of the `Path3D`. Rename the two nodes to `SpawnLocation` and `SpawnPath`, respectively. It's more descriptive of what we'll use them for.



SpawnPath

SpawnLocation

With that, we're ready to code the spawn mechanism.

Spawning monsters randomly

Right-click on the `Main` node and attach a new script to it.

We first export a variable to the *Inspector* so that we can assign `mob.tscn` or any other monster to it.

GDScript

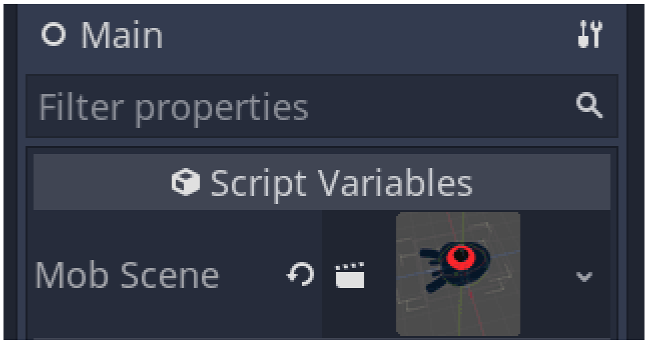
```
extends Node

@export var mob_scene: PackedScene
```

C#

```
using Godot;

public partial class Main : Node
{
    // Don't forget to rebuild the project so the editor knows
about the new export variable.

    [Export]
    public PackedScene MobScene { get; set; }
}
```

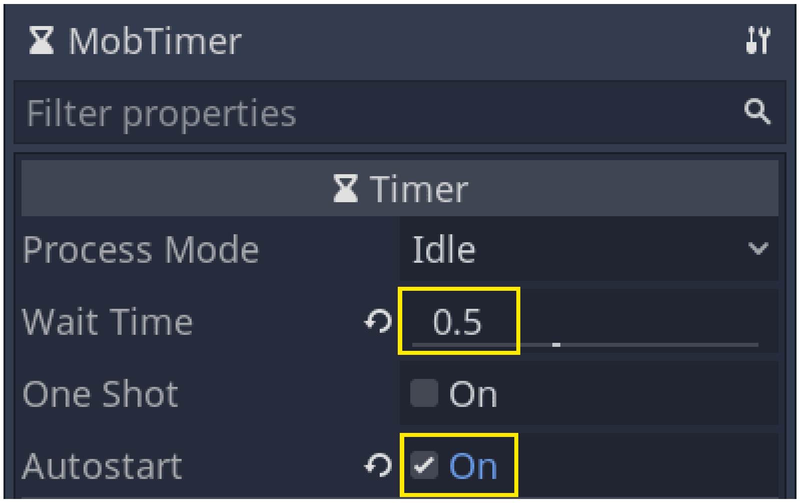
We want to spawn mobs at regular time intervals. To do this, we need to go back to the scene and add a timer. Before that, though, we need to assign the `mob.tscn` file to the `mob_scene` property above (otherwise it's null!)

Head back to the 3D screen and select the `Main` node. Drag `mob.tscn` from the *FileSystem* dock to the *Mob Scene* slot in the *Inspector*.



Add a new [Timer](#) node as a child of `Main`. Name it `MobTimer`.

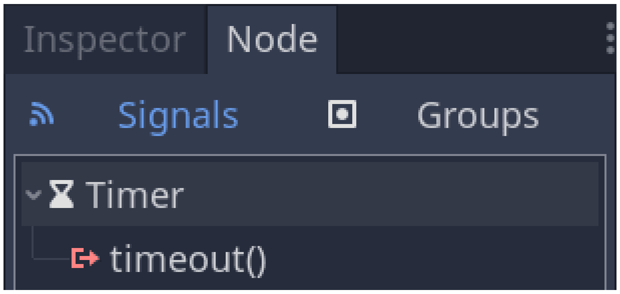
In the *Inspector*, set its *Wait Time* to `0.5` seconds and turn on *Autostart* so it automatically starts when we run the game.


MobTimer
Filter properties
Timer

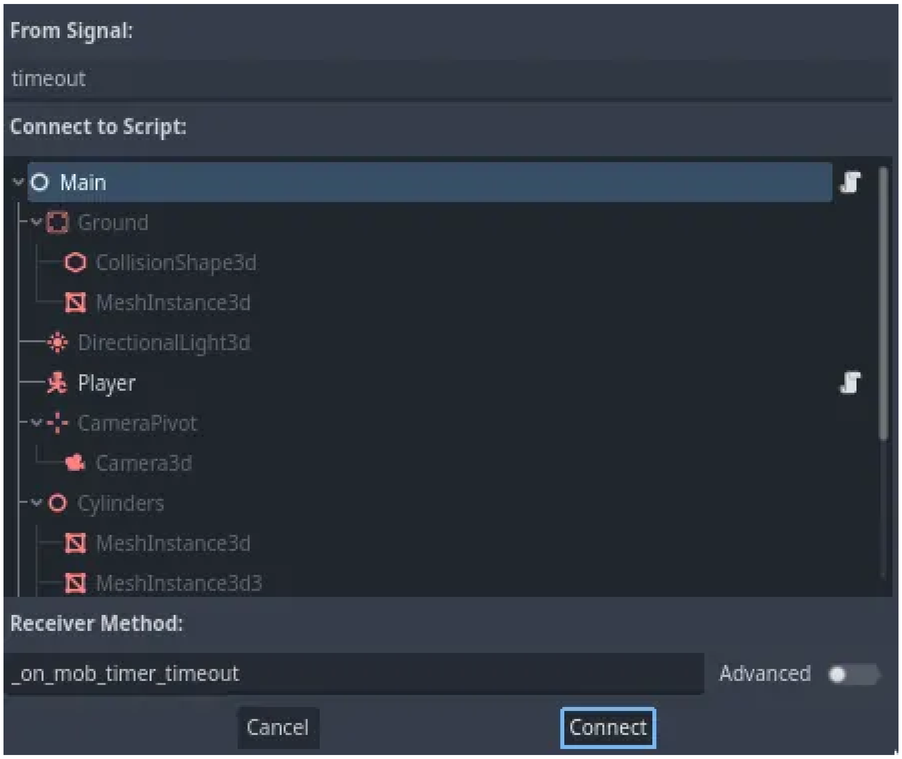
Process Mode: Idle
Wait Time: 0.5
One Shot: ☐ On
Autostart: ☑ On

Timers emit a `timeout` signal every time they reach the end of their *Wait Time*. By default, they restart automatically, emitting the signal in a cycle. We can connect to this signal from the *Main* node to spawn monsters every `0.5` seconds.

With the *MobTimer* still selected, head to the *Node* dock on the right, and double-click the `timeout` signal.


Inspector
Node
Signals
Groups
Timer
timeout()

Connect it to the *Main* node.



From Signal:
timeout
Connect to Script:
Main
Ground
CollisionShape3d
MeshInstance3d
DirectionalLight3d
Player
CameraPivot
Camera3d
Cylinders
MeshInstance3d
MeshInstance3d3
Receiver Method:
_on_mob_timer_timeout
Advanced
Cancel
Connect

This will take you back to the script, with a new empty `_on_mob_timer_timeout()` function.

Let's code the mob spawning logic. We're going to:

1. Instantiate the mob scene.
2. Sample a random position on the spawn path.
3. Get the player's position.
4. Call the mob's `initialize()` method, passing it the random position and the player's position.
5. Add the mob as a child of the *Main* node.

GDScript

```
func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on the SpawnPath.
    # We store the reference to the SpawnLocation node.
    var mob_spawn_location =
get_node("SpawnPath/SpawnLocation")
    # And give it a random offset.
    mob_spawn_location.progress_ratio = randf()

    var player_position = $Player.position
    mob.initialize(mob_spawn_location.position,
player_position)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)
```

C#

```
// We also specified this function name in PascalCase in the
editor's connection window.
private void OnMobTimerTimeout()
{
    // Create a new instance of the Mob scene.
    Mob mob = MobScene.Instantiate<Mob>();

    // Choose a random location on the SpawnPath.
    // We store the reference to the SpawnLocation node.
    var mobSpawnLocation = GetNode<PathFollow3D>
("SpawnPath/SpawnLocation");
    // And give it a random offset.
    mobSpawnLocation.ProgressRatio = GD.Randf();

    Vector3 playerPosition = GetNode<Player>
("Player").Position;
    mob.Initialize(mobSpawnLocation.Position, playerPosition);

    // Spawn the mob by adding it to the Main scene.
    AddChild(mob);
}
```

Above, `randf()` produces a random value between `0` and `1`, which is what the *PathFollow* node's `progress_ratio` expects: 0 is the start of the path, 1 is the end of the path. The path we have set is around the camera's viewport, so any random value between 0 and 1 is a random position alongside the edges of the viewport!

Note that if you remove the `Player` from the main scene, the following line

GDScript

```
var player_position = $Player.position
```

C#

```
Vector3 playerPosition = GetNode<Player>("Player").Position;
```

gives an error because there is no $Player!

Here is the complete `main.gd` script so far, for reference.

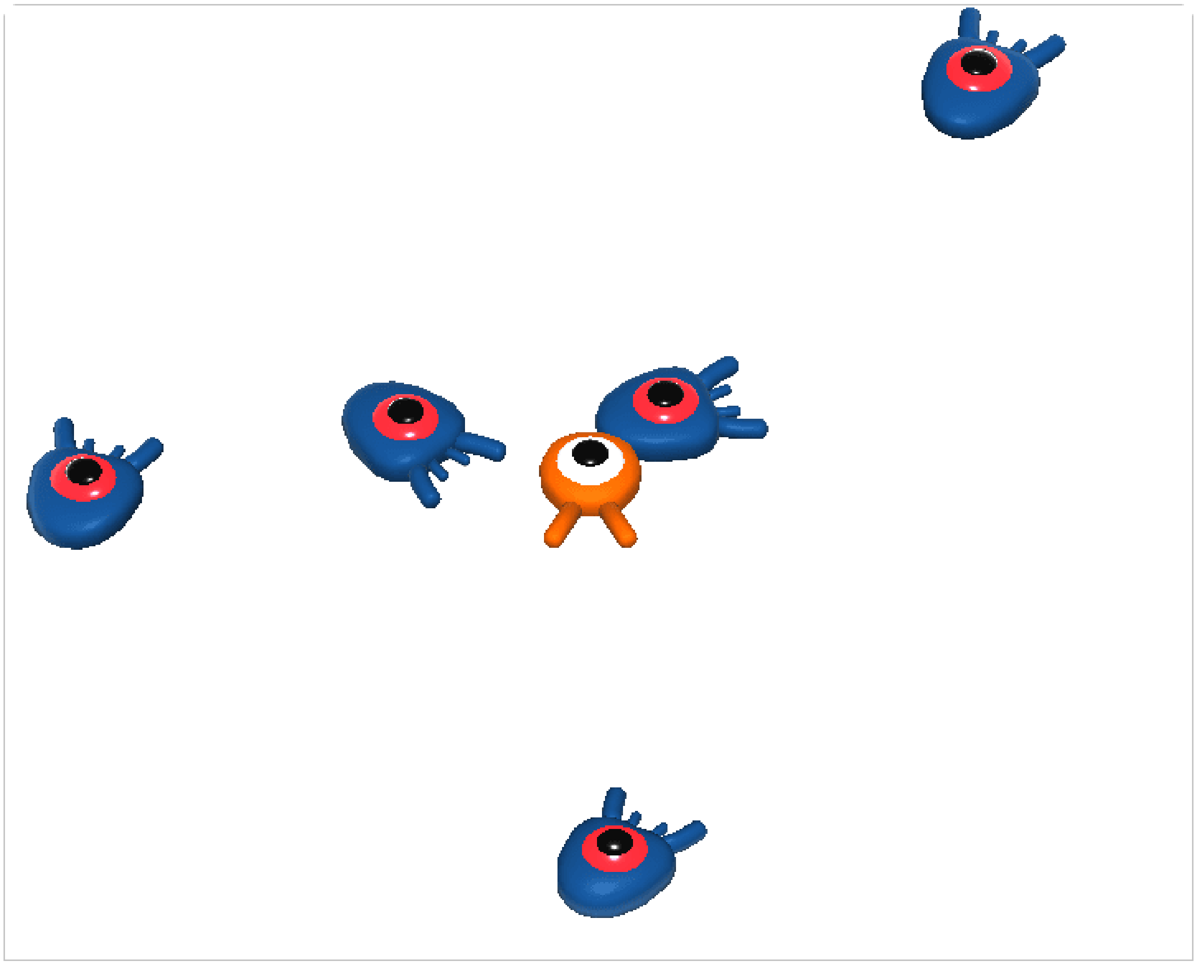
GDScript

```
extends Node

@export var mob_scene: PackedScene


func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on the SpawnPath.
    # We store the reference to the SpawnLocation node.
    var mob_spawn_location =
get_node("SpawnPath/SpawnLocation")
    # And give it a random offset.
    mob_spawn_location.progress_ratio = randf()

    var player_position = $Player.position
    mob.initialize(mob_spawn_location.position,
player_position)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)
```

C#

```
using Godot;

public partial class Main : Node
{
    [Export]
    public PackedScene MobScene { get; set; }

    private void OnMobTimerTimeout()
    {
        // Create a new instance of the Mob scene.
        Mob mob = MobScene.Instantiate<Mob>();

        // Choose a random location on the SpawnPath.
        // We store the reference to the SpawnLocation node.
        var mobSpawnLocation = GetNode<PathFollow3D>
("SpawnPath/SpawnLocation");
        // And give it a random offset.
        mobSpawnLocation.ProgressRatio = GD.Randf();

        Vector3 playerPosition = GetNode<Player>
("Player").Position;
        mob.Initialize(mobSpawnLocation.Position,
playerPosition);

        // Spawn the mob by adding it to the Main scene.
        AddChild(mob);
    }
}
```

You can test the scene by pressing `F6`. You should see the monsters spawn and move in a straight line.



For now, they bump and slide against one another when their paths cross. We'll address this in the next part.

Jumping and squashing monsters

In this part, we'll add the ability to jump and squash the monsters. In the next lesson, we'll make the player die when a monster hits them on the ground.

First, we have to change a few settings related to physics interactions. Enter the world of physics layers.

Controlling physics interactions

Physics bodies have access to two complementary properties: layers and masks. Layers define on which physics layer(s) an object is.

Masks control the layers that a body will listen to and detect. This affects collision detection. When you want two bodies to interact, you need at least one to have a mask corresponding to the other.

If that's confusing, don't worry, we'll see three examples in a second.

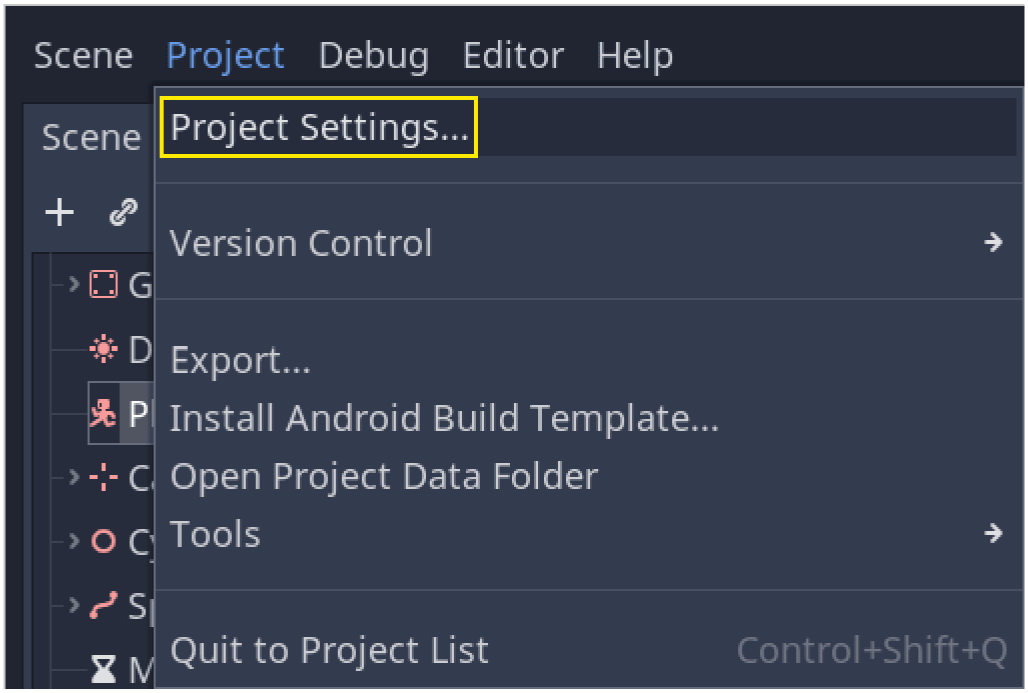
The important point is that you can use layers and masks to filter physics interactions, control performance, and remove the need for extra conditions in your code.

By default, all physics bodies and areas are set to both layer and mask `1`. This means they all collide with each other.

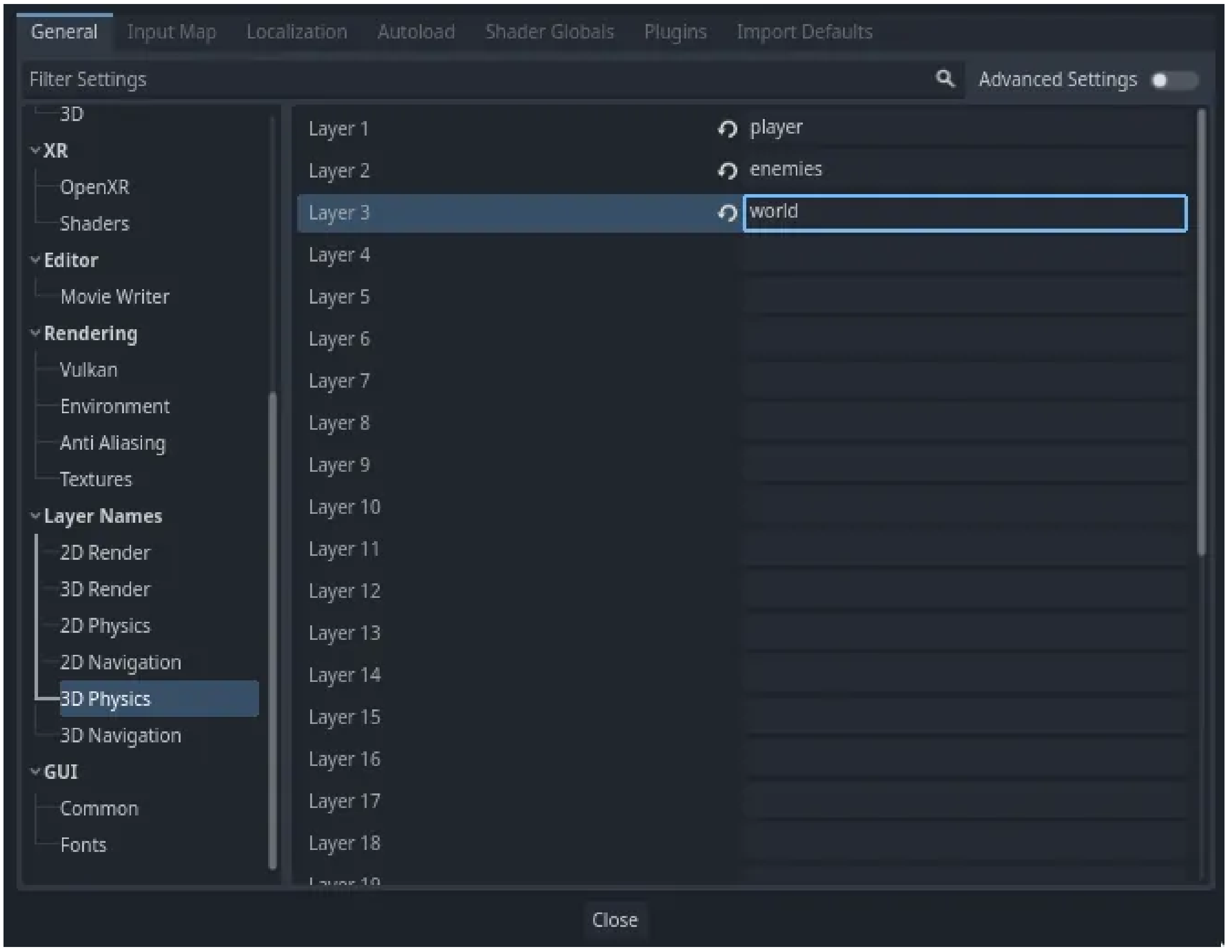
Physics layers are represented by numbers, but we can give them names to keep track of what's what.

Setting layer names

Let's give our physics layers a name. Go to *Project -> Project Settings*.


Scene
Project
Debug
Editor
Help
Scene
Project Settings...
Version Control
Export...
Install Android Build Template...
Open Project Data Folder
Tools
Quit to Project List
Control+Shift+Q

In the left menu, navigate down to *Layer Names -> 3D Physics*. You can see a list of layers with a field next to each of them on the right. You can set their names there. Name the first three layers *player*, *enemies*, and *world*, respectively.


General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Filter Settings
Advanced Settings
3D
XR
OpenXR
Shaders
Editor
Movie Writer
Rendering
Vulkan
Environment
Anti Aliasing
Textures
Layer Names
2D Render
3D Render
2D Physics
2D Navigation
3D Physics
3D Navigation
GUI
Common
Fonts
Layer 4
Layer 5
Layer 6
Layer 7
Layer 8
Layer 9
Layer 10
Layer 11
Layer 12
Layer 13
Layer 14
Layer 15
Layer 16
Layer 17
Layer 18
Close

Layer 1 player

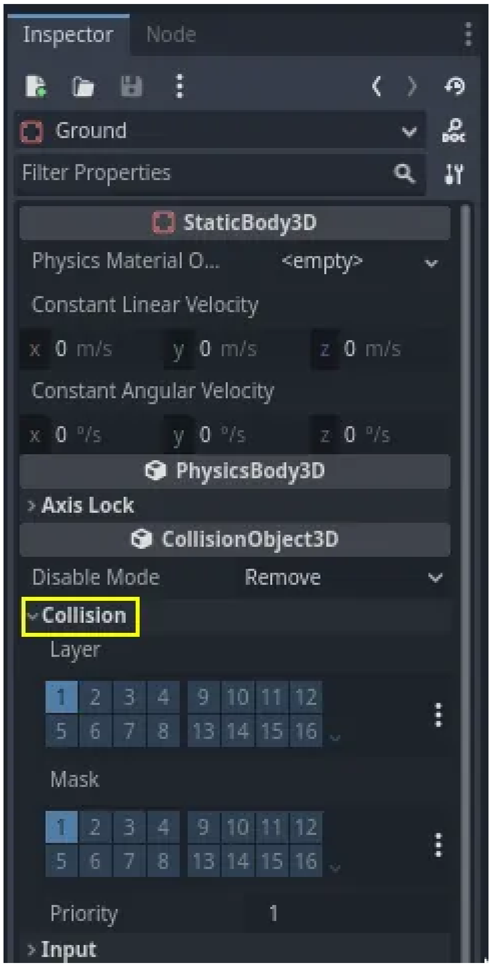
Layer 2 enemies

Layer 3 world

Now, we can assign them to our physics nodes.

Assigning layers and masks

In the *Main* scene, select the `Ground` node. In the *Inspector*, expand the *Collision* section. There, you can see the node's layers and masks as a grid of buttons.


Inspector
Node
Ground
Filter Properties
StaticBody3D
PhysicsBody3D
Axis Lock
CollisionObject3D
Collision
Layer
Mask
Input

Physics Material O... <empty>

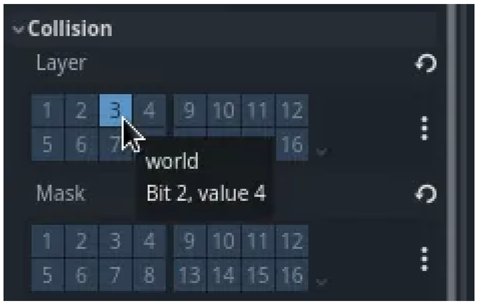
Constant Linear Velocity: x 0 m/s, y 0 m/s, z 0 m/s

Constant Angular Velocity: x 0 °/s, y 0 °/s, z 0 °/s

Disable Mode: Remove

Priority: 1

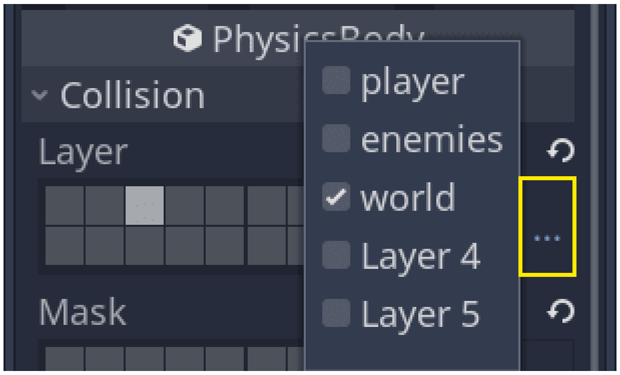
The ground is part of the world, so we want it to be part of the third layer. Click the lit button to toggle **off** the first *Layer* and toggle **on** the third one. Then, toggle **off** the *Mask* by clicking on it.


Collision
Layer
Mask

world
Bit 2, value 4

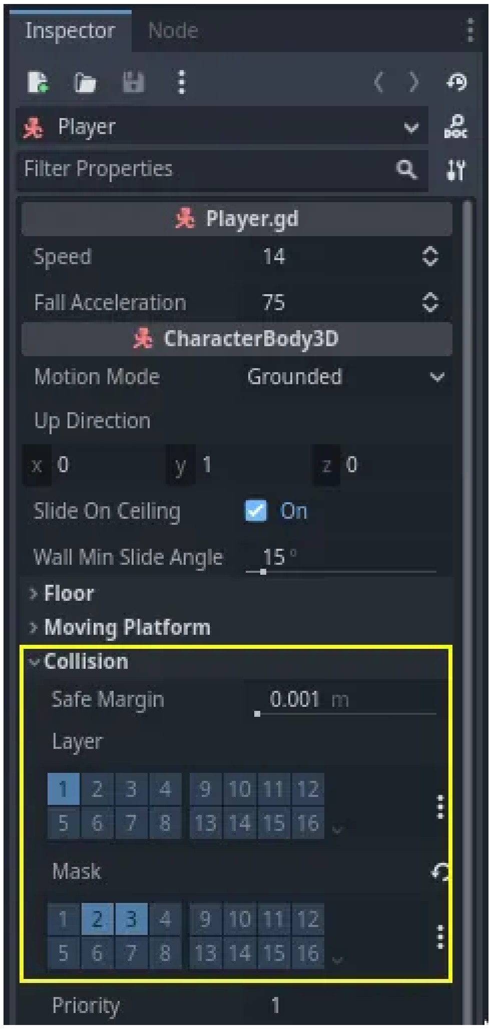
As mentioned before, the *Mask* property allows a node to listen to interaction with other physics objects, but we don't need it to have collisions. `Ground` doesn't need to listen to anything; it's just there to prevent creatures from falling.

Note that you can click the "..." button on the right side of the properties to see a list of named checkboxes.


PhysicsBody
Collision
Layer
Mask
☐ player
☐ enemies
☑ world
☐ Layer 4
☐ Layer 5
...

Next up are the `Player` and the `Mob`. Open `player.tscn` by double-clicking the file in the *FileSystem* dock.

Select the *Player* node and set its *Collision -> Mask* to both "enemies" and "world". You can leave the default *Layer* property as it is, because the first layer is the "player" layer.


Inspector
Node
Player
Filter Properties
Player.gd
CharacterBody3D
Floor
Moving Platform
Collision

Speed 14

Fall Acceleration 75

Motion Mode Grounded

Up Direction x 0 y 1 z 0

Slide On Ceiling ☑ On

Wall Min Slide Angle 15 °

Safe Margin 0.001 m

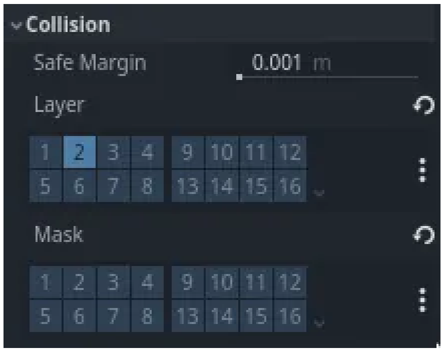
Layer: 1

Mask: 2, 3

Priority 1

Then, open the *Mob* scene by double-clicking on `mob.tscn` and select the `Mob` node.

Set its *Collision -> Layer* to "enemies" and unset its *Collision -> Mask*, leaving the mask empty.



Safe Margin 0.001 m

Layer: 2

Mask:

These settings mean the monsters will move through one another. If you want the monsters to collide with and slide against each other, turn **on** the "enemies" mask.

Note

The mobs don't need to mask the "world" layer because they only move on the XZ plane. We don't apply any gravity to them by design.

Jumping

The jumping mechanic itself requires only two lines of code. Open the *Player* script. We need a value to control the jump's strength and update `_physics_process()` to code the jump.

After the line that defines `fall_acceleration`, at the top of the script, add the `jump_impulse`.

GDScript

```
#...
# Vertical impulse applied to the character upon jumping in
meters per second.
@export var jump_impulse = 20
```

C#

```
// Don't forget to rebuild the project so the editor knows
about the new export variable.

// ...
// Vertical impulse applied to the character upon jumping in
meters per second.
```

```
[Export]
public int JumpImpulse { get; set; } = 20;
```

Inside `_physics_process()`, add the following code before the `move_and_slide()` codeblock.

GDScript

```
func _physics_process(delta):
    #...

    # Jumping.
    if is_on_floor() and Input.is_action_just_pressed("jump"):
        target_velocity.y = jump_impulse

    #...
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...

    // Jumping.
    if (IsOnFloor() && Input.IsActionJustPressed("jump"))
    {
        _targetVelocity.Y = JumpImpulse;
    }

    // ...
}
```

That's all you need to jump!

The `is_on_floor()` method is a tool from the `CharacterBody3D` class. It returns `true` if the body collided with the floor in this frame. That's why we apply gravity to the *Player*: so we collide with the floor instead of floating over it like the monsters.

If the character is on the floor and the player presses "jump", we instantly give them a lot of vertical speed. In games, you really want controls to be responsive and giving instant speed boosts like these, while unrealistic, feels great.

Notice that the Y axis is positive upwards. That's unlike 2D, where the Y axis is positive downwards.

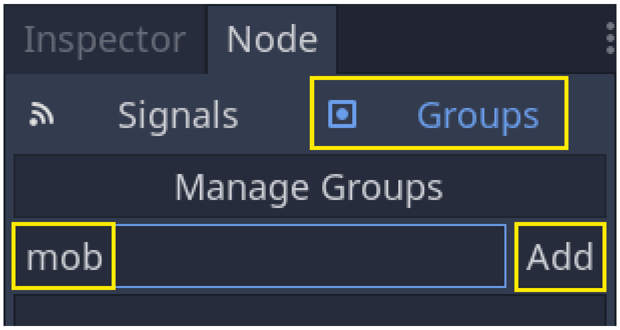
Squashing monsters

Let's add the squash mechanic next. We're going to make the character bounce over monsters and kill them at the same time.

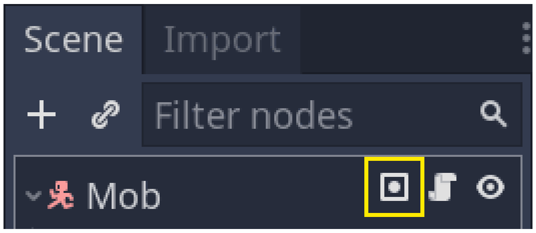
We need to detect collisions with a monster and to differentiate them from collisions with the floor. To do so, we can use Godot's group tagging feature.

Open the scene `mob.tscn` again and select the *Mob* node. Go to the *Node* dock on the right to see a list of signals. The *Node* dock has two tabs: *Signals*, which you've already used, and *Groups*, which allows you to assign tags to nodes.

Click on it to reveal a field where you can write a tag name. Enter "mob" in the field and click the *Add* button.



An icon appears in the *Scene* dock to indicate the node is part of at least one group.



We can now use the group from the code to distinguish collisions with monsters from collisions with the floor.

Coding the squash mechanic

Head back to the *Player* script to code the squash and bounce.

At the top of the script, we need another property, `bounce_impulse`. When squashing an enemy, we don't necessarily want the character to go as high up as when jumping.

GDScript

```
# Vertical impulse applied to the character upon bouncing over a mob in
# meters per second.
@export var bounce_impulse = 16
```

C#

```
// Don't forget to rebuild the project so the editor knows about the new export variable.

// Vertical impulse applied to the character upon bouncing over a mob in meters per second.
[Export]
public int BounceImpulse { get; set; } = 16;
```

Then, after the **Jumping** codeblock we added above in `_physics_process()`, add the following loop. With `move_and_slide()`, Godot makes the body move sometimes multiple times in a row to smooth out the character's motion. So we have to loop over all collisions that may have happened.

In every iteration of the loop, we check if we landed on a mob. If so, we kill it and bounce.

With this code, if no collisions occurred on a given frame, the loop won't run.

GDScript

```
func _physics_process(delta):
    #...

    # Iterate through all collisions that occurred this frame
    for index in range(get_slide_collision_count()):
        # We get one of the collisions with the player
        var collision = get_slide_collision(index)

        # If the collision is with ground
        if collision.get_collider() == null:
            continue

        # If the collider is with a mob
        if collision.get_collider().is_in_group("mob"):
            var mob = collision.get_collider()
            # we check that we are hitting it from above.
            if Vector3.UP.dot(collision.get_normal()) > 0.1:
                # If so, we squash it and bounce.
                mob.squash()
                target_velocity.y = bounce_impulse
                # Prevent further duplicate calls.
                break
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...

    // Iterate through all collisions that occurred this frame.
    for (int index = 0; index < GetSlideCollisionCount();
index++)
    {
        // We get one of the collisions with the player.
        KinematicCollision3D collision =
GetSlideCollision(index);

        // If the collision is with a mob.
        // With C# we leverage typing and pattern-matching
        // instead of checking for the group we created.
        if (collision.GetCollider() is Mob mob)
        {
            // We check that we are hitting it from above.
            if (Vector3.Up.Dot(collision.GetNormal()) > 0.1f)
            {
                // If so, we squash it and bounce.
                mob.Squash();
                _targetVelocity.Y = BounceImpulse;
                // Prevent further duplicate calls.
                break;
            }
        }
    }
}
```

That's a lot of new functions. Here's some more information about them.

The functions `get_slide_collision_count()` and `get_slide_collision()` both come from the CharacterBody3D class and are related to `move_and_slide()`.

`get_slide_collision()` returns a KinematicCollision3D object that holds information about where and how the collision occurred. For example, we use its `get_collider` property to check if we collided with a "mob" by calling `is_in_group()` on it: `collision.get_collider().is_in_group("mob")`.

Note

The method `is_in_group()` is available on every Node.

To check that we are landing on the monster, we use the vector dot product: `Vector3.UP.dot(collision.get_normal()) > 0.1`. The collision normal is a 3D vector that is perpendicular to the plane where the collision occurred. The dot product allows us to compare it to the up direction.

With dot products, when the result is greater than `0`, the two vectors are at an angle of fewer than 90 degrees. A value higher than `0.1` tells us that we are roughly above the monster.

After handling the squash and bounce logic, we terminate the loop early via the `break` statement to prevent further duplicate calls to `mob.squash()`, which may otherwise result in unintended bugs such as counting the score multiple times for one kill.

We are calling one undefined function, `mob.squash()`, so we have to add it to the Mob class.

Open the script `mob.gd` by double-clicking on it in the *FileSystem* dock. At the top of the script, we want to define a new signal named `squashed`. And at the bottom, you can add the squash function, where we emit the signal and destroy the mob.

GDScript

```
# Emitted when the player jumped on the mob.
signal squashed

# ...


func squash():
    squashed.emit()
    queue_free()
```

C#

```
// Don't forget to rebuild the project so the editor knows
about the new signal.

// Emitted when the player jumped on the mob.
[Signal]
public delegate void SquashedEventHandler();

// ...

public void Squash()
{
    EmitSignal(SignalName.Squashed);
    QueueFree();
}
```

Note

When using C#, Godot will create the appropriate events automatically for all Signals ending with *EventHandler*, see C# Signals.

We will use the signal to add points to the score in the next lesson.

With that, you should be able to kill monsters by jumping on them. You can press `F5` to try the game and set `main.tscn` as your project's main scene.

However, the player won't die yet. We'll work on that in the next part.

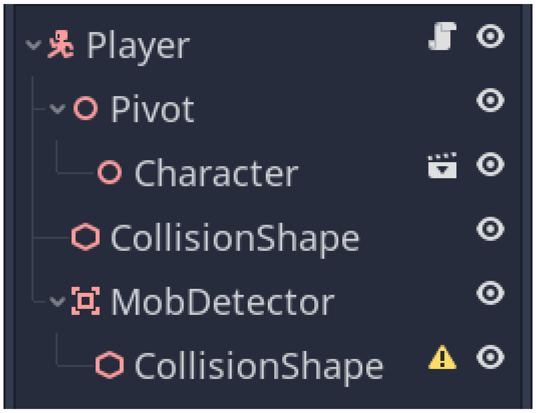
Killing the player

We can kill enemies by jumping on them, but the player still can't die. Let's fix this.

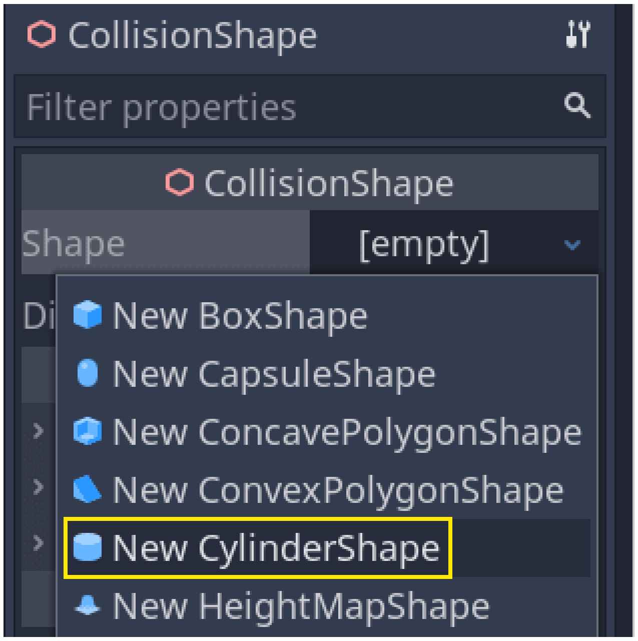
We want to detect being hit by an enemy differently from squashing them. We want the player to die when they're moving on the floor, but not if they're in the air. We could use vector math to distinguish the two kinds of collisions. Instead, though, we will use an Area3D node, which works well for hitboxes.

Hitbox with the Area node

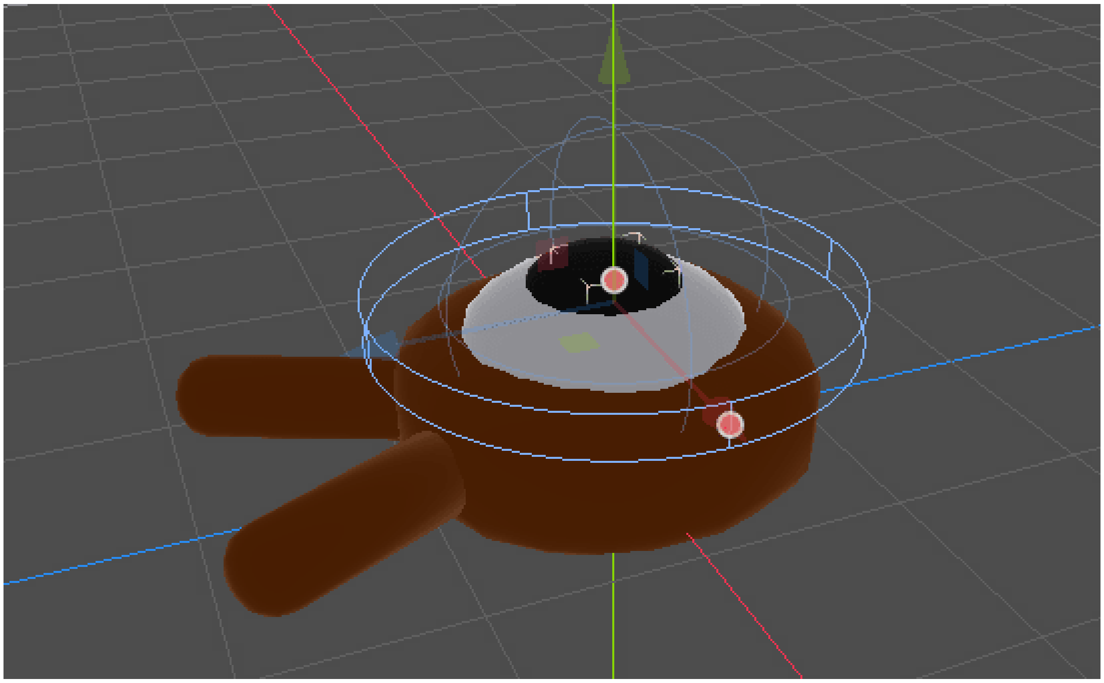
Head back to the `player.tscn` scene and add a new child node Area3D. Name it `MobDetector` Add a CollisionShape3D node as a child of it.


Player
Pivot
Character
CollisionShape
MobDetector
CollisionShape

In the *Inspector*, assign a cylinder shape to it.


CollisionShape
Filter properties
CollisionShape
Shape
[empty]
Di
New BoxShape
New CapsuleShape
New ConcavePolygonShape
New ConvexPolygonShape
New CylinderShape
New HeightMapShape

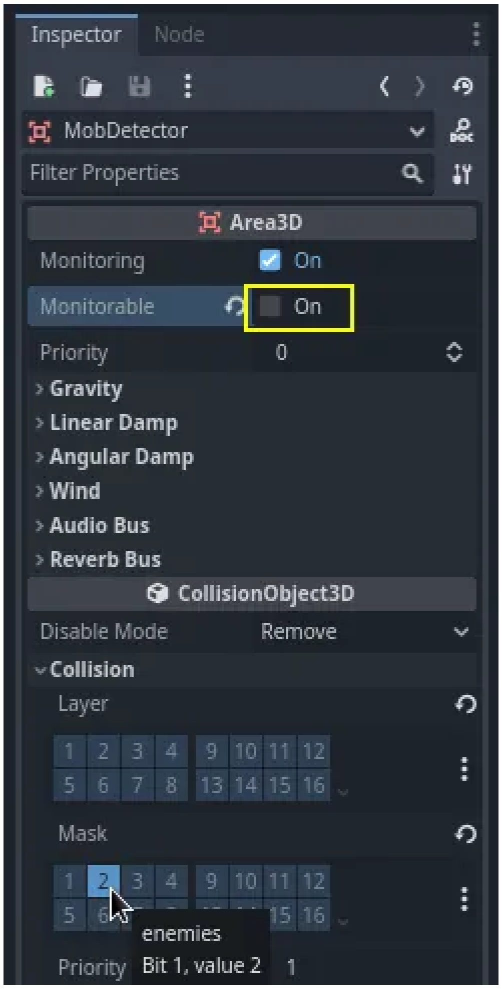
Here is a trick you can use to make the collisions only happen when the player is on the ground or close to it. You can reduce the cylinder's height and move it up to the top of the character. This way, when the player jumps, the shape will be too high up for the enemies to collide with it.



You also want the cylinder to be wider than the sphere. This way, the player gets hit before colliding and being pushed on top of the monster's collision box.

The wider the cylinder, the more easily the player will get killed.

Next, select the `MobDetector` node again, and in the *Inspector*, turn **off** its *Monitorable* property. This makes it so other physics nodes cannot detect the area. The complementary *Monitoring* property allows it to detect collisions. Then, remove the *Collision -> Layer* and set the mask to the "enemies" layer.



MobDetector

Area3D

Monitoring ☑ On

Monitorable ☐ On

Priority 0

CollisionObject3D

Disable Mode Remove

Collision

Layer

Mask 2

enemies

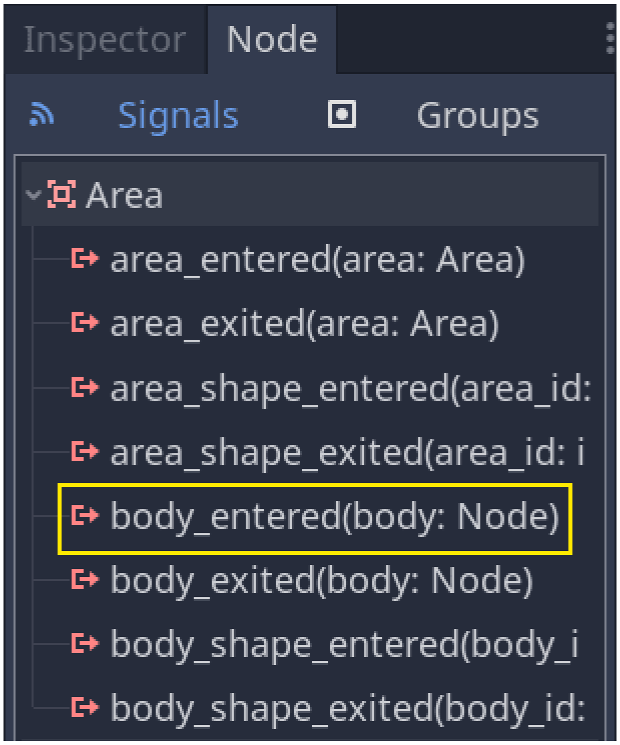
Bit 1, value 2

Priority 1

Inspector
Node
Filter Properties
Gravity
Linear Damp
Angular Damp
Wind
Audio Bus
Reverb Bus
1 2 3 4 9 10 11 12
5 6 7 8 13 14 15 16

When areas detect a collision, they emit signals. We're going to connect one to the `Player` node. Select `MobDetector` and go to

Inspector's *Node* tab, double-click the `body_entered` signal and connect it to the `Player`


Inspector
Node
Signals
Groups
Area
area_entered(area: Area)
area_exited(area: Area)
area_shape_entered(area_id:
area_shape_exited(area_id: i
body_entered(body: Node)
body_exited(body: Node)
body_shape_entered(body_i
body_shape_exited(body_id:

The *MobDetector* will emit `body_entered` when a CharacterBody3D or a RigidBody3D node enters it. As it only masks the "enemies" physics layers, it will only detect the `Mob` nodes.

Code-wise, we're going to do two things: emit a signal we'll later use to end the game and destroy the player. We can wrap these operations in a `die()` function that helps us put a descriptive label on the code.

GDScript

```
# Emitted when the player was hit by a mob.
# Put this at the top of the script.
signal hit


# And this function at the bottom.
func die():
    hit.emit()
    queue_free()


func _on_mob_detector_body_entered(body):
    die()
```

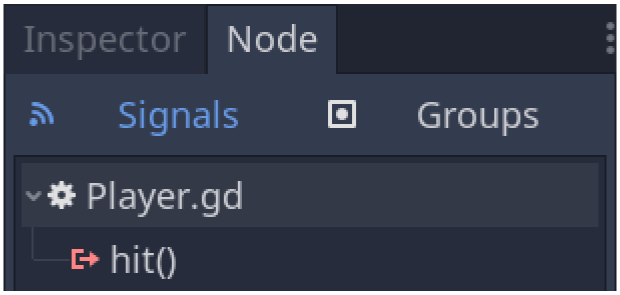
C#

```
// Don't forget to rebuild the project so the editor knows
about the new signal.

// Emitted when the player was hit by a mob.
[Signal]
public delegate void HitEventHandler();

// ...

private void Die()
{
    EmitSignal(SignalName.Hit);
    QueueFree();
}

// We also specified this function name in PascalCase in the
editor's connection window.
private void OnMobDetectorBodyEntered(Node3D body)
{
    Die();
}
```

Ending the game

We can use the `Player`'s `hit` signal to end the game. All we need to do is connect it to the `Main` node and stop the `MobTimer` in reaction.

Open `main.tscn`, select the `Player` node, and in the *Node* dock, connect its `hit` signal to the `Main` node.



Get the timer, and stop it, in the `_on_player_hit()` function.

GDScript

```
func _on_player_hit():
    $MobTimer.stop()
```

C#

```
// We also specified this function name in PascalCase in the
editor's connection window.
private void OnPlayerHit()
{
    GetNode<Timer>("MobTimer").Stop();
}
```

If you try the game now, the monsters will stop spawning when you die, and the remaining ones will leave the screen.

Notice also that the game no longer crashes or displays an error when the player dies. Because we are stopping the MobTimer, it no longer triggers the `_on_mob_timer_timeout()` function.

Also note that the enemy colliding with the player and dying depends on the size and position of the `Player` and the `Mob`'s collision shapes. You may need to move them and resize them to achieve a tight game feel.

You can pat yourself on the back: you prototyped a complete 3D game, even if it's still a bit rough.

From there, we'll add a score, the option to retry the game, and you'll see how you can make the game feel much more alive with minimalistic animations.

Code checkpoint

Here are the complete scripts for the `Main`, `Mob`, and `Player` nodes, for reference. You can use them to compare and check your code.

Starting with `main.gd`.

GDScript

```
extends Node

@export var mob_scene: PackedScene


func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on the SpawnPath.
    # We store the reference to the SpawnLocation node.
    var mob_spawn_location =
get_node("SpawnPath/SpawnLocation")
    # And give it a random offset.
    mob_spawn_location.progress_ratio = randf()

    var player_position = $Player.position
    mob.initialize(mob_spawn_location.position,
player_position)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)

func _on_player_hit():
    $MobTimer.stop()
```

C#

```
using Godot;

public partial class Main : Node
{
    [Export]
    public PackedScene MobScene { get; set; }

    private void OnMobTimerTimeout()
    {
        // Create a new instance of the Mob scene.
        Mob mob = MobScene.Instantiate<Mob>();

        // Choose a random location on the SpawnPath.
        // We store the reference to the SpawnLocation node.
        var mobSpawnLocation = GetNode<PathFollow3D>
("SpawnPath/SpawnLocation");
        // And give it a random offset.
        mobSpawnLocation.ProgressRatio = GD.Randf();

        Vector3 playerPosition = GetNode<Player>
("Player").Position;
        mob.Initialize(mobSpawnLocation.Position,
playerPosition);

        // Spawn the mob by adding it to the Main scene.
        AddChild(mob);
    }

    private void OnPlayerHit()
    {
        GetNode<Timer>("MobTimer").Stop();
    }
}
```

Next is `mob.gd`.

GDScript

```
extends CharacterBody3D
```

```
# Minimum speed of the mob in meters per second.
@export var min_speed = 10
# Maximum speed of the mob in meters per second.
@export var max_speed = 18

# Emitted when the player jumped on the mob
signal squashed

func _physics_process(_delta):
    move_and_slide()

# This function will be called from the Main scene.
func initialize(start_position, player_position):
    # We position the mob by placing it at start_position
    # and rotate it towards player_position, so it looks at the player.
    look_at_from_position(start_position, player_position, Vector3.UP)
    # Rotate this mob randomly within range of -45 and +45 degrees,
    # so that it doesn't move directly towards the player.
    rotate_y(randf_range(-PI / 4, PI / 4))

    # We calculate a random speed (integer)
    var random_speed = randi_range(min_speed, max_speed)
    # We calculate a forward velocity that represents the speed.
    velocity = Vector3.FORWARD * random_speed
    # We then rotate the velocity vector based on the mob's Y rotation
    # in order to move in the direction the mob is looking.
    velocity = velocity.rotated(Vector3.UP, rotation.y)

func _on_visible_on_screen_notifier_3d_screen_exited():
    queue_free()

func squash():
    squashed.emit()
    queue_free() # Destroy this node
```

C#

```
using Godot;

public partial class Mob : CharacterBody3D
{
    // Emitted when the played jumped on the mob.
    [Signal]
    public delegate void SquashedEventHandler();

    // Minimum speed of the mob in meters per second
    [Export]
    public int MinSpeed { get; set; } = 10;
    // Maximum speed of the mob in meters per second
    [Export]
    public int MaxSpeed { get; set; } = 18;

    public override void _PhysicsProcess(double delta)
    {
        MoveAndSlide();
    }

    // This function will be called from the Main scene.
    public void Initialize(Vector3 startPosition, Vector3
playerPosition)
    {
        // We position the mob by placing it at startPosition
        // and rotate it towards playerPosition, so it looks at
the player.
        LookAtFromPosition(startPosition, playerPosition,
Vector3.Up);
        // Rotate this mob randomly within range of -45 and +45
degrees,
        // so that it doesn't move directly towards the player.
        RotateY((float)GD.RandRange(-Mathf.Pi / 4.0, Mathf.Pi /
4.0));

        // We calculate a random speed (integer)
        int randomSpeed = GD.RandRange(MinSpeed, MaxSpeed);
        // We calculate a forward velocity that represents the
speed.
        Velocity = Vector3.Forward * randomSpeed;
        // We then rotate the velocity vector based on the
mob's Y rotation
        // in order to move in the direction the mob is
looking.
        Velocity = Velocity.Rotated(Vector3.Up, Rotation.Y);
    }

    public void Squash()
    {
        EmitSignal(SignalName.Squashed);
        QueueFree(); // Destroy this node
    }

    private void OnVisibilityNotifierScreenExited()
    {
        QueueFree();
    }
}
```

Finally, the longest script, `player.gd`:

GDScript

```
extends CharacterBody3D

signal hit

# How fast the player moves in meters per second
@export var speed = 14
# The downward acceleration while in the air, in meters per
second squared.
@export var fall_acceleration = 75
# Vertical impulse applied to the character upon jumping in
meters per second.
@export var jump_impulse = 20
# Vertical impulse applied to the character upon bouncing over
a mob
# in meters per second.
@export var bounce_impulse = 16

var target_velocity = Vector3.ZERO


func _physics_process(delta):
    # We create a local variable to store the input direction
    var direction = Vector3.ZERO

    # We check for each move input and update the direction
accordingly
    if Input.is_action_pressed("move_right"):
        direction.x = direction.x + 1
    if Input.is_action_pressed("move_left"):
        direction.x = direction.x - 1
    if Input.is_action_pressed("move_back"):
        # Notice how we are working with the vector's x and z
axes.
        # In 3D, the XZ plane is the ground plane.
        direction.z = direction.z + 1
    if Input.is_action_pressed("move_forward"):
        direction.z = direction.z - 1

    # Prevent diagonal moving fast af
    if direction != Vector3.ZERO:
        direction = direction.normalized()
        $Pivot.look_at(position + direction, Vector3.UP)

    # Ground Velocity
    target_velocity.x = direction.x * speed
    target_velocity.z = direction.z * speed

    # Vertical Velocity
    if not is_on_floor(): # If in the air, fall towards the
floor. Literally gravity
        target_velocity.y = target_velocity.y -
(fall_acceleration * delta)

    # Jumping.
    if is_on_floor() and Input.is_action_just_pressed("jump"):
        target_velocity.y = jump_impulse

    # Iterate through all collisions that occurred this frame
    # in C this would be for(int i = 0; i < collisions.Count;
i++)
    for index in range(get_slide_collision_count()):
        # We get one of the collisions with the player
        var collision = get_slide_collision(index)
        # If the collision is with ground
        if collision.get_collider() == null:
            continue

        # If the collider is with a mob
        if collision.get_collider().is_in_group("mob"):
            var mob = collision.get_collider()
            # we check that we are hitting it from above.
            if Vector3.UP.dot(collision.get_normal()) > 0.1:
                # If so, we squash it and bounce.
                mob.squash()
                target_velocity.y = bounce_impulse
                # Prevent further duplicate calls.
                break

    # Moving the Character
    velocity = target_velocity
    move_and_slide()

# And this function at the bottom.
func die():
    hit.emit()
    queue_free()

func _on_mob_detector_body_entered(body):
    die()
```

C#

```
using Godot;

public partial class Player : CharacterBody3D
{
    // Emitted when the player was hit by a mob.
    [Signal]
    public delegate void HitEventHandler();

    // How fast the player moves in meters per second.
    [Export]
    public int Speed { get; set; } = 14;
    // The downward acceleration when in the air, in meters per
second squared.
```

```
    [Export]
    public int FallAcceleration { get; set; } = 75;
    // Vertical impulse applied to the character upon jumping in meters per second.
    [Export]
    public int JumpImpulse { get; set; } = 20;
    // Vertical impulse applied to the character upon bouncing over a mob in meters per second.
    [Export]
    public int BounceImpulse { get; set; } = 16;

    private Vector3 _targetVelocity = Vector3.Zero;

    public override void _PhysicsProcess(double delta)
    {
        // We create a local variable to store the input direction.
        var direction = Vector3.Zero;

        // We check for each move input and update the direction accordingly.
        if (Input.IsActionPressed("move_right"))
        {
            direction.X += 1.0f;
        }
        if (Input.IsActionPressed("move_left"))
        {
            direction.X -= 1.0f;
        }
        if (Input.IsActionPressed("move_back"))
        {
            // Notice how we are working with the vector's X and Z axes.
            // In 3D, the XZ plane is the ground plane.
            direction.Z += 1.0f;
        }
        if (Input.IsActionPressed("move_forward"))
        {
            direction.Z -= 1.0f;
        }

        // Prevent diagonal moving fast af
        if (direction != Vector3.Zero)
        {
            direction = direction.Normalized();
            GetNode<Node3D>("Pivot").LookAt(Position +
direction, Vector3.Up);
        }

        // Ground Velocity
        _targetVelocity.X = direction.X * Speed;
        _targetVelocity.Z = direction.Z * Speed;

        // Vertical Velocity
        if (!IsOnFloor()) // If in the air, fall towards the
floor. Literally gravity
        {
            _targetVelocity.Y -= FallAcceleration *
(float)delta;
        }

        // Jumping.
        if (IsOnFloor() && Input.IsActionJustPressed("jump"))
        {
            _targetVelocity.Y = JumpImpulse;
        }

        // Iterate through all collisions that occurred this
frame.
        for (int index = 0; index < GetSlideCollisionCount();
index++)
        {
            // We get one of the collisions with the player.
            KinematicCollision3D collision =
GetSlideCollision(index);

            // If the collision is with a mob.
            if (collision.GetCollider() is Mob mob)
            {
                // We check that we are hitting it from above.
                if (Vector3.Up.Dot(collision.GetNormal()) >
0.1f)
                {
                    // If so, we squash it and bounce.
                    mob.Squash();
                    _targetVelocity.Y = BounceImpulse;
                    // Prevent further duplicate calls.
                    break;
                }
            }
        }

        // Moving the Character
        Velocity = _targetVelocity;
        MoveAndSlide();
    }

    private void Die()
    {
        EmitSignal(SignalName.Hit);
        QueueFree();
    }

    private void OnMobDetectorBodyEntered(Node3D body)
    {
        Die();
    }
}
```

See you in the next lesson to add the score and the retry option.

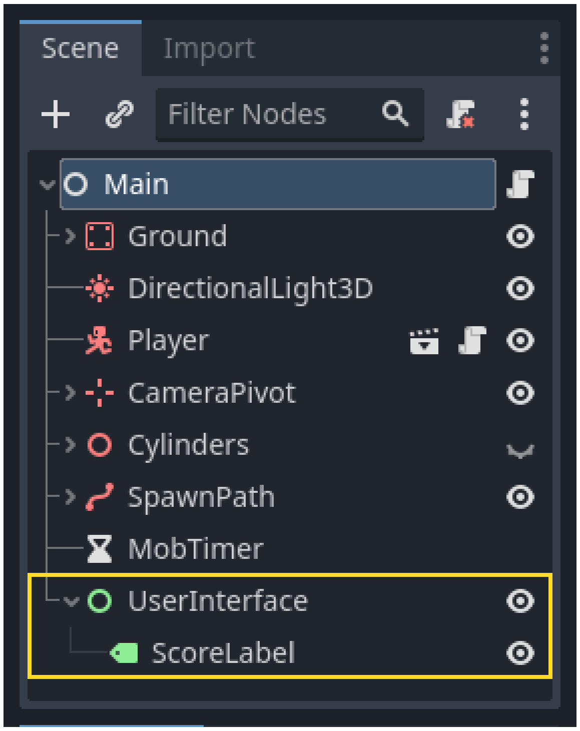
Score and replay

In this part, we'll add the score, music playback, and the ability to restart the game.

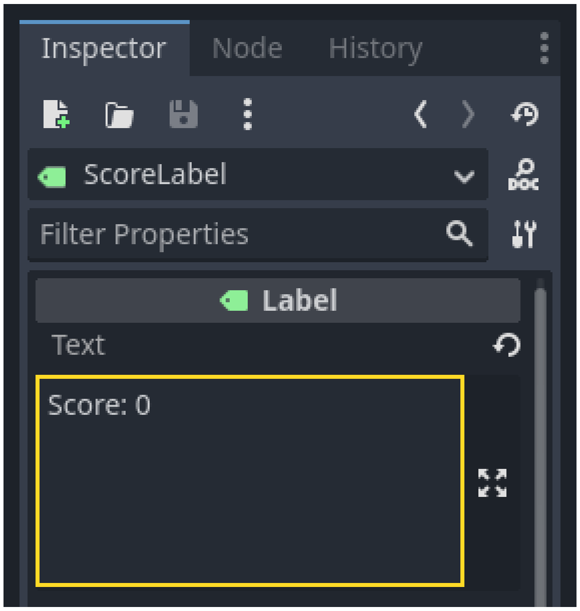
We have to keep track of the current score in a variable and display it on screen using a minimal interface. We will use a text label to do that.

In the main scene, add a new child node Control to `Main` and name it `UserInterface`. Ensure you are on the 2D screen, where you can edit your User Interface (UI).

Add a Label node and name it `ScoreLabel`


Scene
Import
Filter Nodes
Main
Ground
DirectionalLight3D
Player
CameraPivot
Cylinders
SpawnPath
MobTimer
UserInterface
ScoreLabel

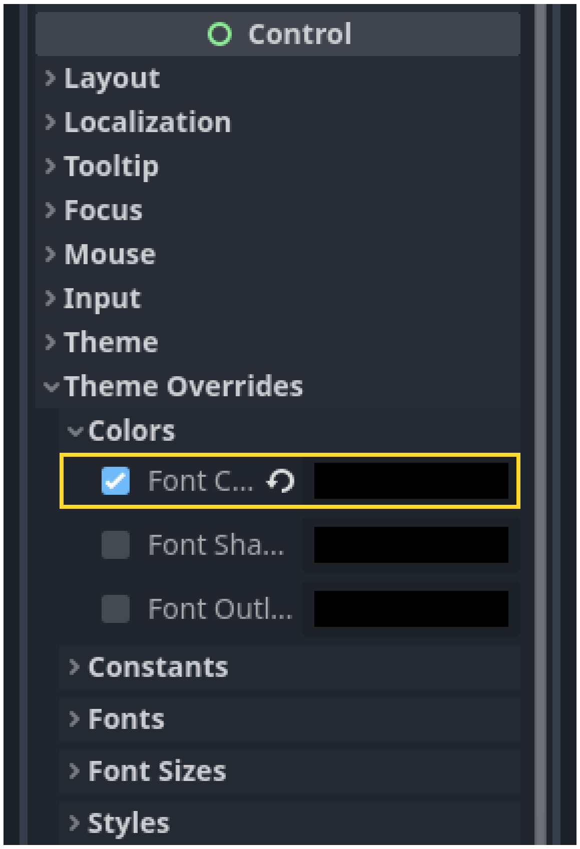
In the *Inspector*, set the *Label*'s *Text* to a placeholder like "Score: 0".


Inspector
Node
History
ScoreLabel
Filter Properties
Label
Text

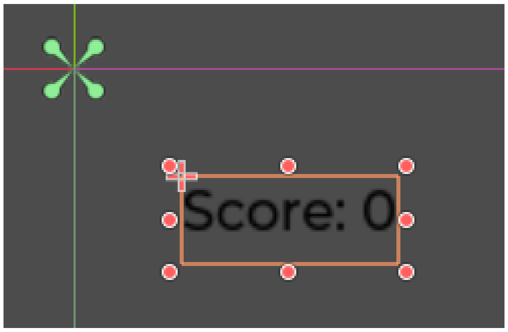
Score: 0

Also, the text is white by default, like our game's background. We need to change its color to see it at runtime.

Scroll down to *Theme Overrides*, and expand *Colors* and enable *Font Color* in order to tint the text to black (which contrasts well with the white 3D scene)


Control
Layout
Localization
Tooltip
Focus
Mouse
Input
Theme
Theme Overrides
Colors
Font C...
Font Sha...
Font Outl...
Constants
Fonts
Font Sizes
Styles

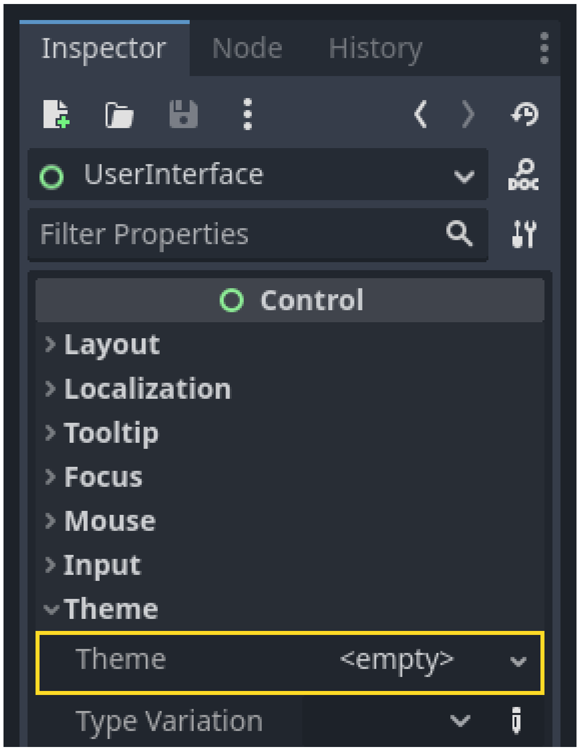
Finally, click and drag on the text in the viewport to move it away from the top-left corner.



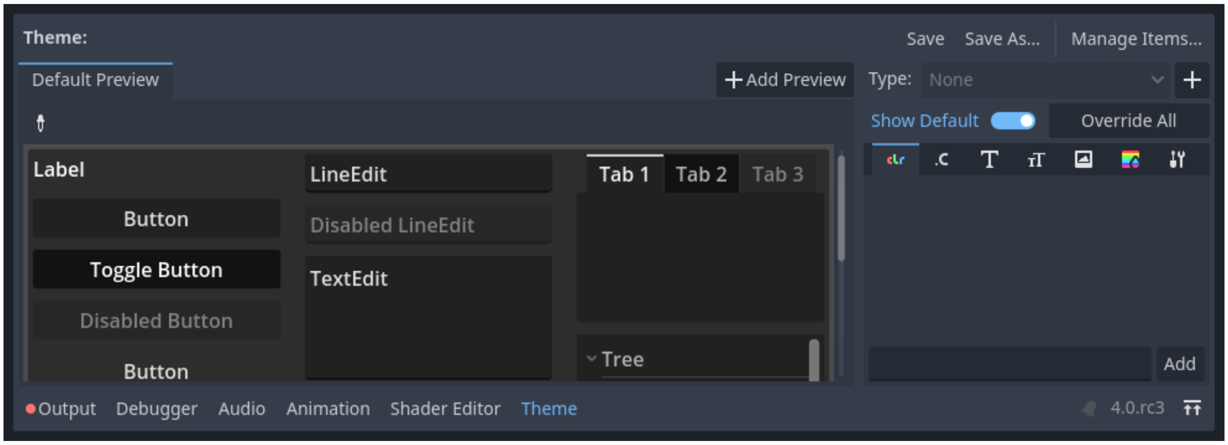
The `UserInterface` node allows us to group our UI in a branch of the scene tree and use a theme resource that will propagate to all its children. We'll use it to set our game's font.

Creating a UI theme

Once again, select the `UserInterface` node. In the *Inspector*, create a new theme resource in *Theme -> Theme*.


Inspector
Node
History
UserInterface
Filter Properties
Control
Layout
Localization
Tooltip
Focus
Mouse
Input
Theme
Theme <empty>
Type Variation

Click on it to open the theme editor In the bottom panel. It gives you a preview of how all the built-in UI widgets will look with your theme resource.


Theme:
Save
Save As...
Manage Items...
Default Preview
+Add Preview
Type: None
Show Default
Override All
Label
Button
Toggle Button
Disabled Button
Button
LineEdit
Disabled LineEdit
TextEdit
Tab 1
Tab 2
Tab 3
Tree
Add
Output
Debugger
Audio
Animation
Shader Editor
Theme
4.0.rc3

By default, a theme only has a few properties: *Default Base Scale*, *Default Font* and *Default Font Size*.

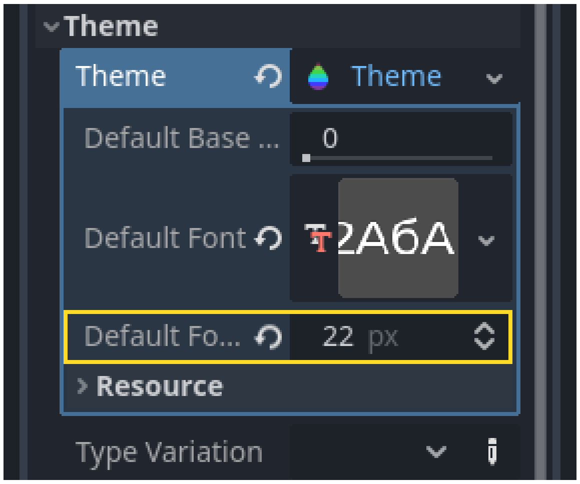
See also

You can add more properties to the theme resource to design complex user interfaces, but that is beyond the scope of this series. To learn more about creating and editing themes, see Introduction to GUI skinning.

The *Default Font* expects a font file like the ones you have on your computer. Two common font file formats are TrueType Font (TTF) and OpenType Font (OTF).

In the *FileSystem* dock, expand the `fonts` directory and click and drag the `Montserrat-Medium.ttf` file we included in the project onto the *Default Font*. The text will reappear in the theme preview.

The text is a bit small. Set the *Default Font Size* to `22` pixels to increase the text's size.



Theme: Theme
Default Base ...: 0
Default Font: 2A6A
Default Fo...: 22 px

Keeping track of the score

Let's work on the score next. Attach a new script to the `ScoreLabel` and define the `score` variable.

GDScript

```
extends Label

var score = 0
```

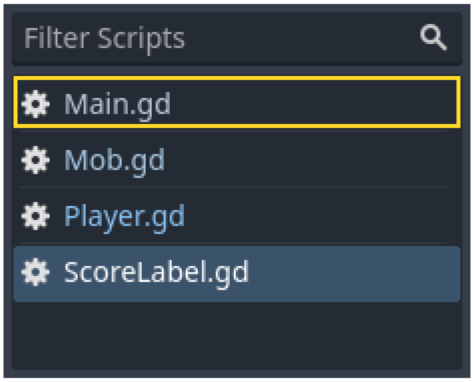
C#

```
using Godot;

public partial class ScoreLabel : Label
{
    private int _score = 0;
}
```

The score should increase by `1` every time we squash a monster. We can use their `squashed` signal to know when that happens. However, because we instantiate monsters from the code, we cannot connect the mob signal to the `ScoreLabel` via the editor.

Instead, we have to make the connection from the code every time we spawn a monster.

Open the script `main.gd`. If it's still open, you can click on its name in the script editor's left column.



Alternatively, you can double-click the `main.gd` file in the *FileSystem* dock.

At the bottom of the `_on_mob_timer_timeout()` function, add the following line:

GDScript

```
func _on_mob_timer_timeout():
    #...
    # We connect the mob to the score label to update the score
upon squashing one.

mob.squashed.connect($UserInterface/ScoreLabel._on_mob_squashed
.bind())
```

C#

```
private void OnMobTimerTimeout()
{
    // ...
    // We connect the mob to the score label to update the
score upon squashing one.
    mob.Squashed += GetNode<ScoreLabel>
("UserInterface/ScoreLabel").OnMobSquashed;
}
```

This line means that when the mob emits the `squashed` signal, the `ScoreLabel` node will receive it and call the function `_on_mob_squashed()`.

Head back to the `score_label.gd` script to define the `_on_mob_squashed()` callback function.

There, we increment the score and update the displayed text.

GDScript

```
func _on_mob_squashed():
    score += 1
    text = "Score: %s" % score
```

C#

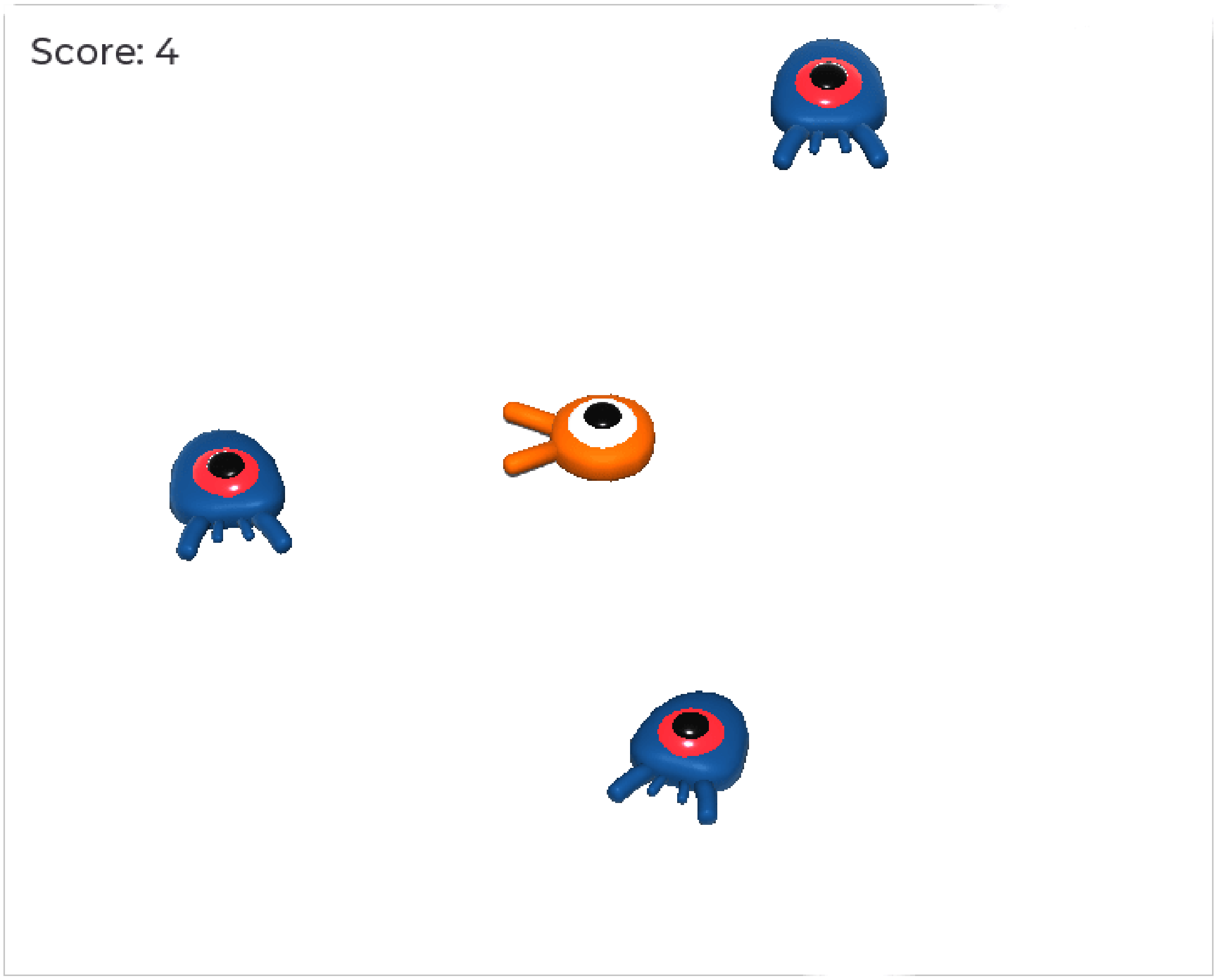
```
public void OnMobSquashed()
{
    _score += 1;
    Text = $"Score: {_score}";
}
```

The second line uses the value of the `score` variable to replace the placeholder `%s`. When using this feature, Godot automatically converts values to string text, which is convenient when outputting text in labels or when using the `print()` function.

See also

You can learn more about string formatting here: GDScript format strings. In C#, consider using string interpolation with "$" [https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated].

You can now play the game and squash a few enemies to see the score increase.



Note

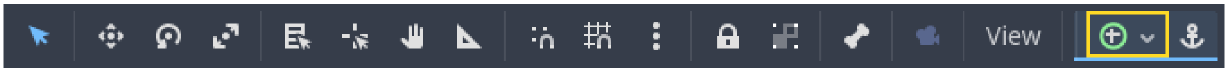
In a complex game, you may want to completely separate your user interface from the game world. In that case, you would not keep track of the score on the label. Instead, you may want to store it in a separate, dedicated object. But when prototyping or when your project is simple, it is fine to keep your code simple. Programming is always a balancing act.

Retrying the game

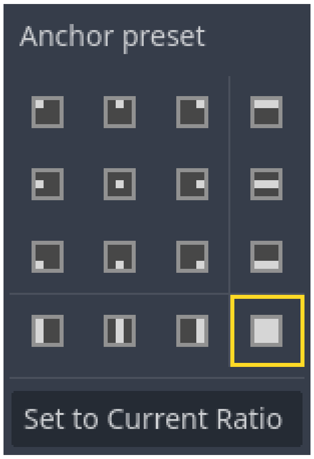
We'll now add the ability to play again after dying. When the player dies, we'll display a message on the screen and wait for input.

Head back to the `main.tscn` scene, select the `UserInterface` node, add a child node ColorRect, and name it `Retry`. This node fills a rectangle with a uniform color and will serve as an overlay to darken the screen.

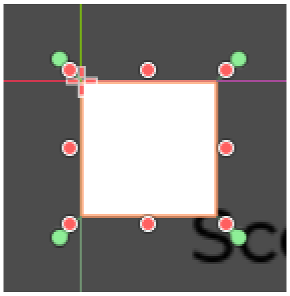
To make it span over the whole viewport, you can use the *Anchor Preset* menu in the toolbar.



Open it and apply the *Full Rect* command.



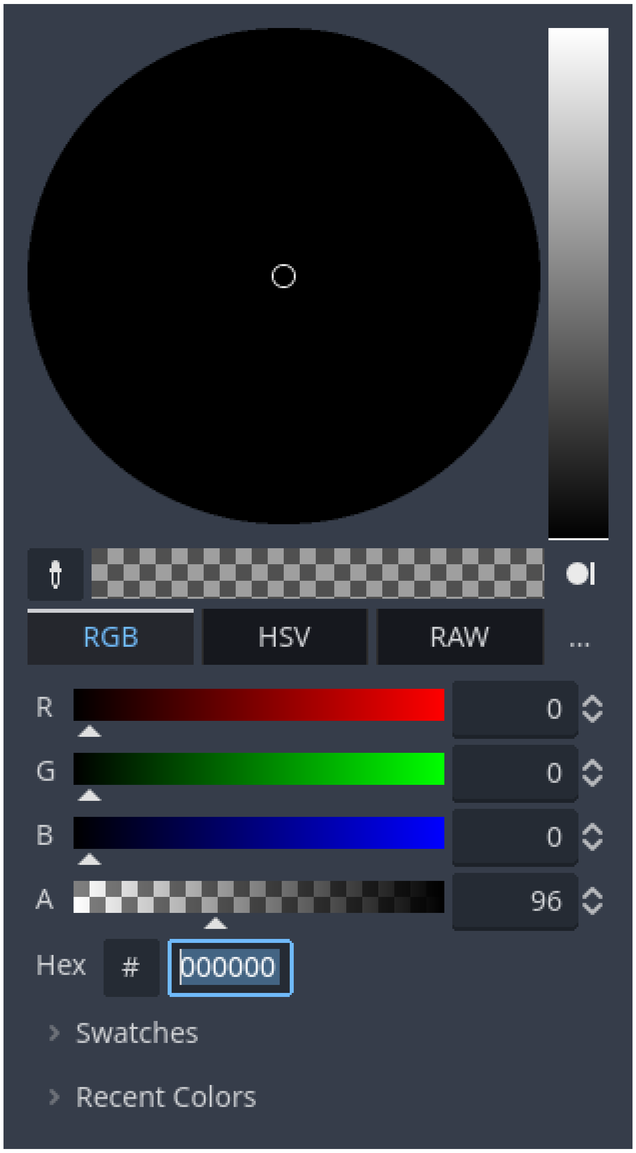
Nothing happens. Well, almost nothing; only the four green pins move to the corners of the selection box.



This is because UI nodes (all the ones with a green icon) work with anchors and margins relative to their parent's bounding box. Here, the `UserInterface` node has a small size and the `Retry` one is limited by it.

Select the `UserInterface` and apply *Anchor Preset -> Full Rect* to it as well. The `Retry` node should now span the whole viewport.

Let's change its color so it darkens the game area. Select `Retry` and in the *Inspector*, set its *Color* to something both dark and transparent. To do so, in the color picker, drag the *A* slider to the left. It controls the color's Alpha channel, that is to say, its opacity/transparency.


RGB
HSV
RAW
...
Swatches
Recent Colors

R 0

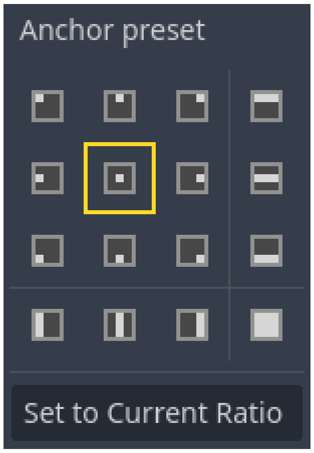
G 0

B 0

A 96

Hex # 000000

Next, add a Label as a child of `Retry` and give it the *Text* "Press Enter to retry." To move it and anchor it in the center of the screen, apply *Anchor Preset -> Center* to it.



Coding the retry option

We can now head to the code to show and hide the `Retry` node when the player dies and plays again.

Open the script `main.gd`. First, we want to hide the overlay at the start of the game. Add this line to the `_ready()` function.

GDScript

```
func _ready():
    $UserInterface/Retry.hide()
```

C#

```
public override void _Ready()
{
    GetNode<Control>("UserInterface/Retry").Hide();
}
```

Then, when the player gets hit, we show the overlay.

GDScript

```
func _on_player_hit():
    #...
    $UserInterface/Retry.show()
```

C#

```
private void OnPlayerHit()
{
    //...
    GetNode<Control>("UserInterface/Retry").Show();
}
```

Finally, when the `Retry` node is visible, we need to listen to the player's input and restart the game if they press enter. To do this, we use the built-in `_unhandled_input()` callback, which is triggered on any input.

If the player pressed the predefined `ui_accept` input action and `Retry` is visible, we reload the current scene.

GDScript

```
func _unhandled_input(event):
    if event.is_action_pressed("ui_accept") and
$UserInterface/Retry.visible:
        # This restarts the current scene.
        get_tree().reload_current_scene()
```

C#

```
public override void _UnhandledInput(InputEvent @event)
{
    if (@event.IsActionPressed("ui_accept") && GetNode<Control>
("UserInterface/Retry").Visible)
    {
        // This restarts the current scene.
        GetTree().ReloadCurrentScene();
    }
}
```

The function `get_tree()` gives us access to the global SceneTree object, which allows us to reload and restart the current scene.

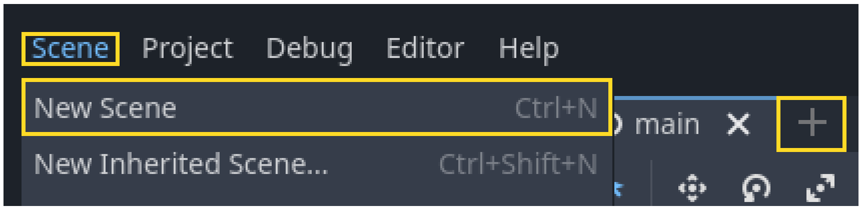
Adding music

To add music that plays continuously in the background, we're going to use another feature in Godot: autoloads.

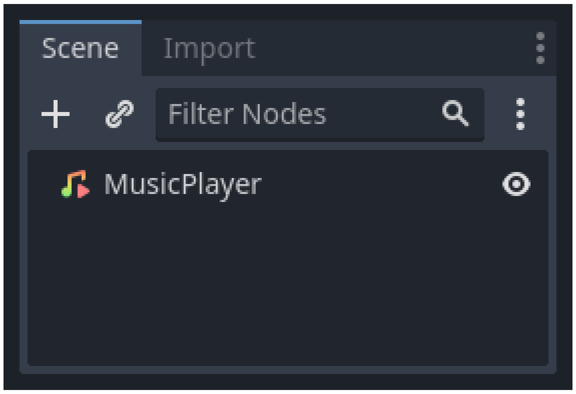
To play audio, all you need to do is add an AudioStreamPlayer node to your scene and attach an audio file to it. When you start the scene, it can play automatically. However, when you reload the scene, like we do to play again, the audio nodes are also reset, and the music starts back from the beginning.

You can use the autoload feature to have Godot load a node or a scene automatically at the start of the game, outside the current scene. You can also use it to create globally accessible objects.

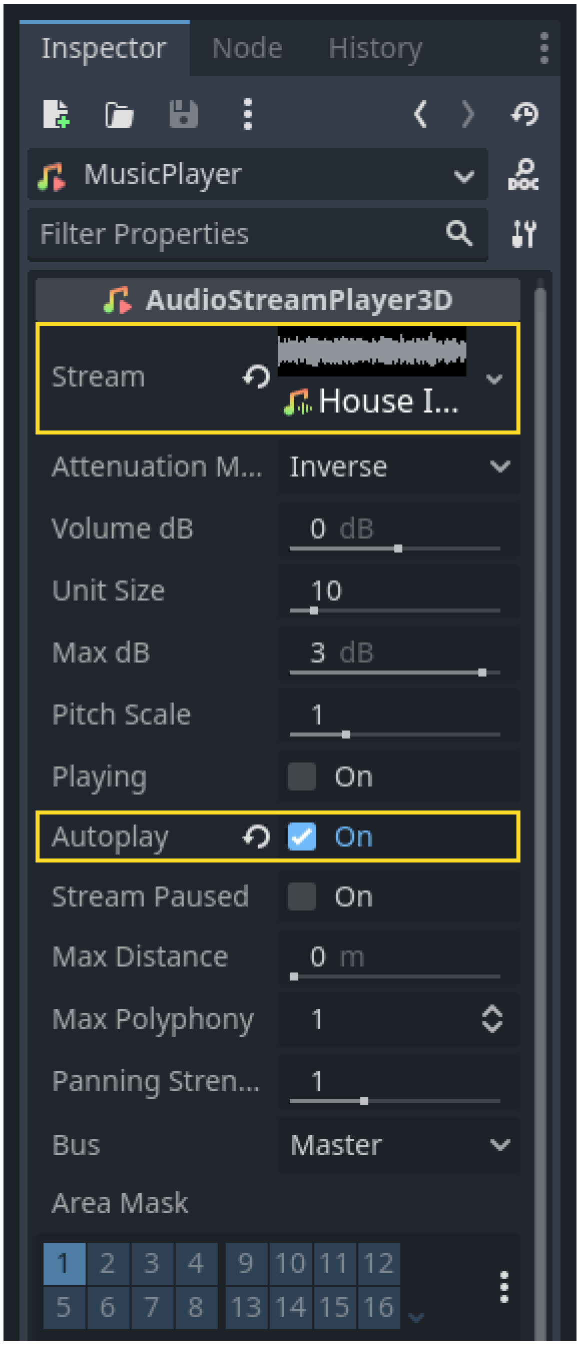
Create a new scene by going to the *Scene* menu and clicking *New Scene* or by using the *+* icon next to your currently opened scene.


Scene
Project
Debug
Editor
Help
New Scene
Ctrl+N
New Inherited Scene...
Ctrl+Shift+N
main

Click the *Other Node* button to create an AudioStreamPlayer and rename it to `MusicPlayer`.


Scene
Import
Filter Nodes
MusicPlayer

We included a music soundtrack in the `art/` directory, `House In a Forest Loop.ogg`. Click and drag it onto the *Stream* property in the *Inspector*. Also, turn on *Autoplay* so the music plays automatically at the start of the game.

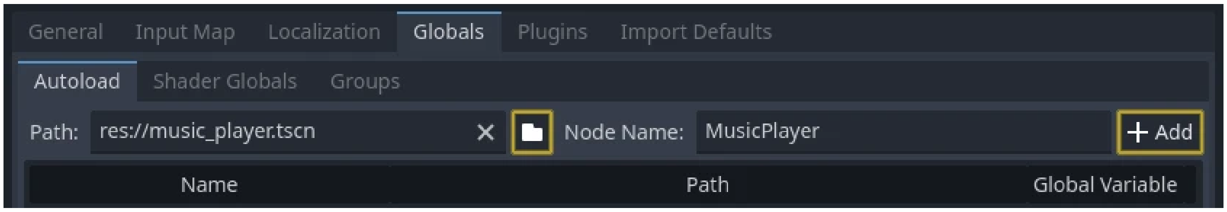

Inspector
Node
History
MusicPlayer
Filter Properties
AudioStreamPlayer3D

Stream: House I...
Attenuation M...: Inverse
Volume dB: 0 dB
Unit Size: 10
Max dB: 3 dB
Pitch Scale: 1
Playing: ☐ On
Autoplay: ☑ On
Stream Paused: ☐ On
Max Distance: 0 m
Max Polyphony: 1
Panning Stren...: 1
Bus: Master
Area Mask: 1 2 3 4 9 10 11 12 5 6 7 8 13 14 15 16

Save the scene as `music_player.tscn`.

We have to register it as an autoload. Head to the *Project -> Project Settings…* menu and click on the *Globals -> Autoload* tab.

In the *Path* field, you want to enter the path to your scene. Click the folder icon to open the file browser and double-click on `music_player.tscn`. Then, click the *Add* button on the right to register the node.

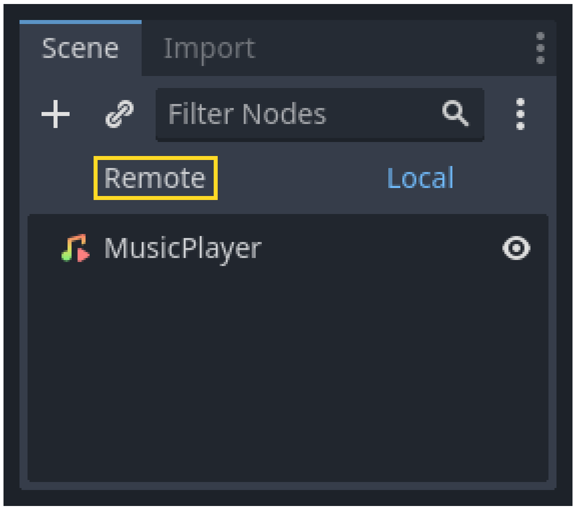

General
Input Map
Localization
Globals
Plugins
Import Defaults
Autoload
Shader Globals
Groups

Path: res://music_player.tscn
Node Name: MusicPlayer

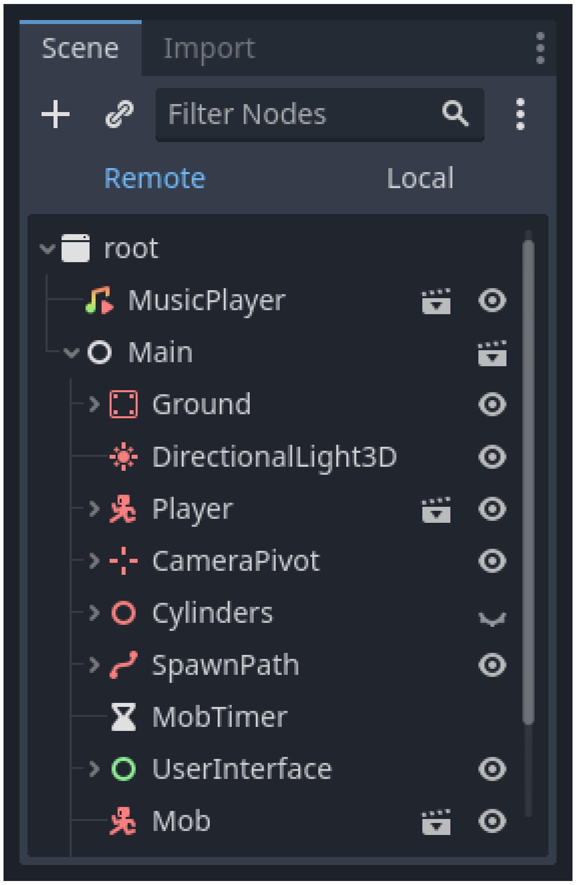
+ Add
Name
Path
Global Variable

`music_player.tscn` now loads into any scene you open or play. So if you run the game now, the music will play automatically in any scene.

Before we wrap up this lesson, here's a quick look at how it works under the hood. When you run the game, your *Scene* dock changes to give you two tabs: *Remote* and *Local*.


Scene
Import
Filter Nodes
Remote
Local
MusicPlayer

The *Remote* tab allows you to visualize the node tree of your running game. There, you will see the *Main* node and everything the scene contains and the instantiated mobs at the bottom.


Scene
Import
Filter Nodes
Remote
Local
root
MusicPlayer
Main
Ground
DirectionalLight3D
Player
CameraPivot
Cylinders
SpawnPath
MobTimer
UserInterface
Mob

At the top are the autoloaded `MusicPlayer` and a *root* node, which is your game's viewport.

And that does it for this lesson. In the next part, we'll add an animation to make the game both look and feel much nicer.

Here is the complete `main.gd` script for reference.

GDScript

```
extends Node

@export var mob_scene: PackedScene

func _ready():
    $UserInterface/Retry.hide()
```

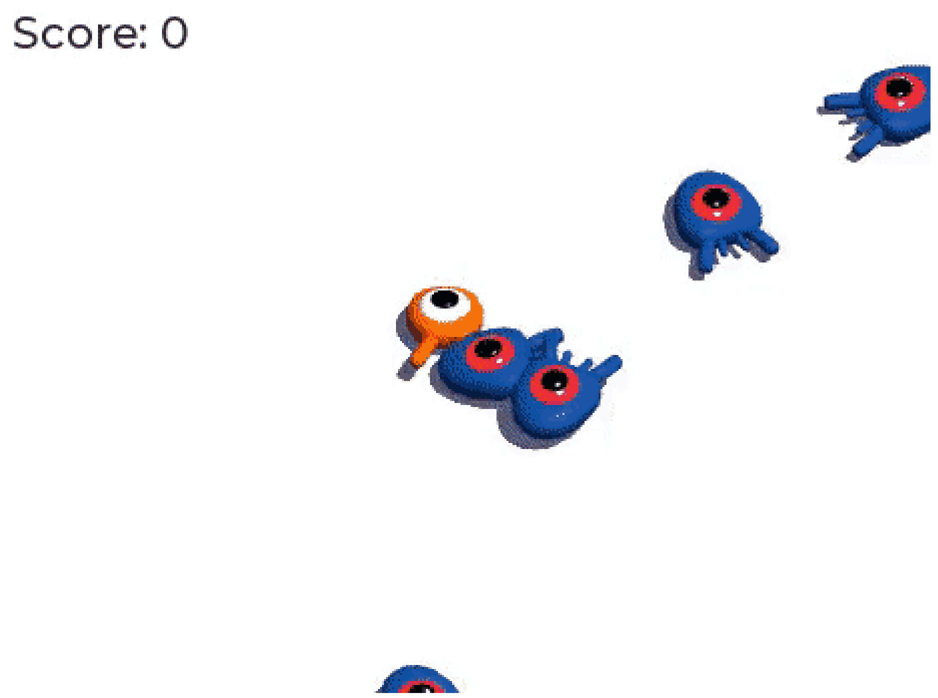
```
func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on the SpawnPath.
    # We store the reference to the SpawnLocation node.
    var mob_spawn_location =
get_node("SpawnPath/SpawnLocation")
    # And give it a random offset.
    mob_spawn_location.progress_ratio = randf()

    var player_position = $Player.position
    mob.initialize(mob_spawn_location.position,
player_position)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)

    # We connect the mob to the score label to update the score
upon squashing one.

mob.squashed.connect($UserInterface/ScoreLabel._on_mob_squashed
.bind())

func _on_player_hit():
    $MobTimer.stop()
    $UserInterface/Retry.show()

func _unhandled_input(event):
    if event.is_action_pressed("ui_accept") and
$UserInterface/Retry.visible:
        # This restarts the current scene.
        get_tree().reload_current_scene()
```

C#

```
using Godot;

public partial class Main : Node
{
    [Export]
    public PackedScene MobScene { get; set; }
    public override void _Ready()
    {
        GetNode<Control>("UserInterface/Retry").Hide();
    }

    public override void _UnhandledInput(InputEvent @event)
    {
        if (@event.IsActionPressed("ui_accept") &&
GetNode<Control>("UserInterface/Retry").Visible)
        {
            // This restarts the current scene.
            GetTree().ReloadCurrentScene();
        }
    }

    private void OnMobTimerTimeout()
    {
        // Create a new instance of the Mob scene.
        Mob mob = MobScene.Instantiate<Mob>();

        // Choose a random location on the SpawnPath.
        // We store the reference to the SpawnLocation node.
        var mobSpawnLocation = GetNode<PathFollow3D>
("SpawnPath/SpawnLocation");
        // And give it a random offset.
        mobSpawnLocation.ProgressRatio = GD.Randf();

        Vector3 playerPosition = GetNode<Player>
("Player").position;
        mob.Initialize(mobSpawnLocation.Position,
playerPosition);

        // Spawn the mob by adding it to the Main scene.
        AddChild(mob);

        // We connect the mob to the score label to update the
score upon squashing one.
        mob.Squashed += GetNode<ScoreLabel>
("UserInterface/ScoreLabel").OnMobSquashed;
    }

    private void OnPlayerHit()
    {
        GetNode<Timer>("MobTimer").Stop();
        GetNode<Control>("UserInterface/Retry").Show();
    }
}
```

Character animation

In this final lesson, we'll use Godot's built-in animation tools to make our characters float and flap. You'll learn to design animations in the editor and use code to make your game feel alive.



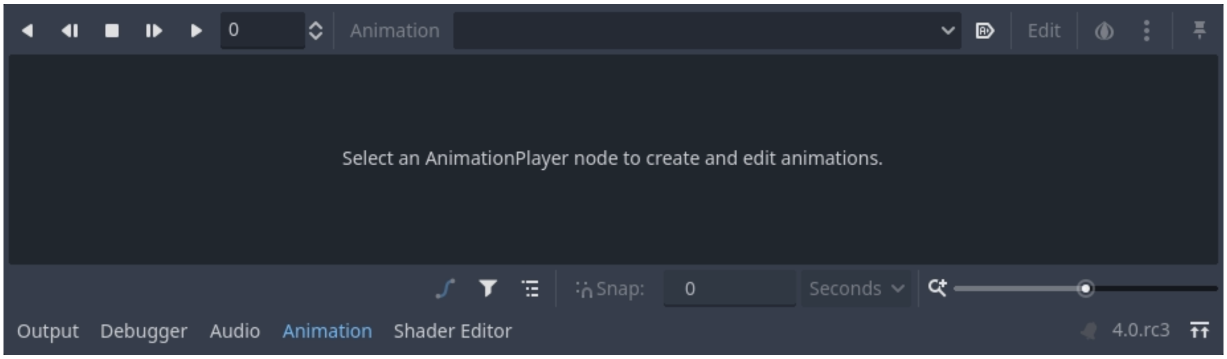
We'll start with an introduction to using the animation editor.

Using the animation editor

The engine comes with tools to author animations in the editor. You can then use the code to play and control them at runtime.

Open the player scene, select the `Player` node, and add an AnimationPlayer node.

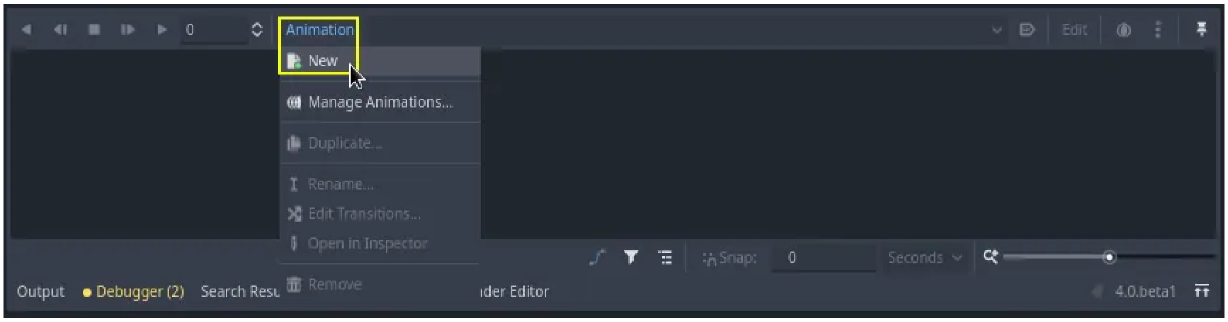
The *Animation* dock appears in the bottom panel.


Select an AnimationPlayer node to create and edit animations.

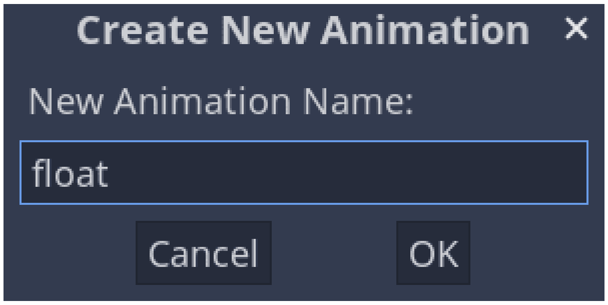
0
Animation
Edit
Snap:
0
Seconds
Output
Debugger
Audio
Animation
Shader Editor
4.0.rc3

It features a toolbar and the animation drop-down menu at the top, a track editor in the middle that's currently empty, and filter, snap, and zoom options at the bottom.

Let's create an animation. Click on *Animation -> New*.

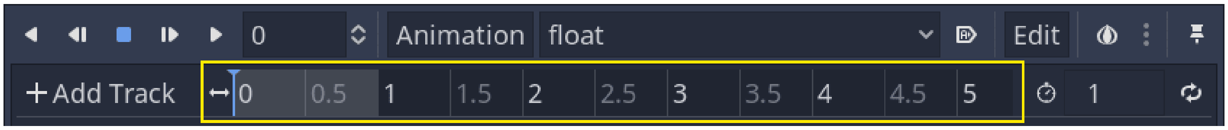

0
Animation
New
Manage Animations...
Duplicate...
Rename...
Edit Transitions...
Open in Inspector
Remove
Edit
Snap:
0
Seconds
Output
Debugger (2)
Search Resu
ıder Editor
4.0.beta1

Name the animation "float".


Create New Animation

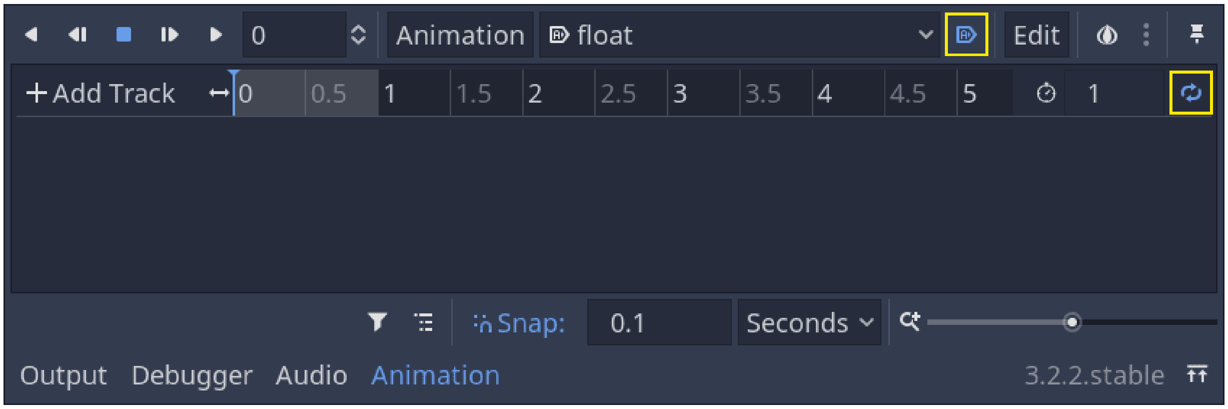
New Animation Name: float

Once you've created the animation, the timeline appears with numbers representing time in seconds.

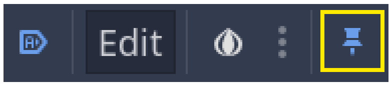

0
Animation
float
Edit
+Add Track
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
1

We want the animation to start playback automatically at the start of the game. Also, it should loop.

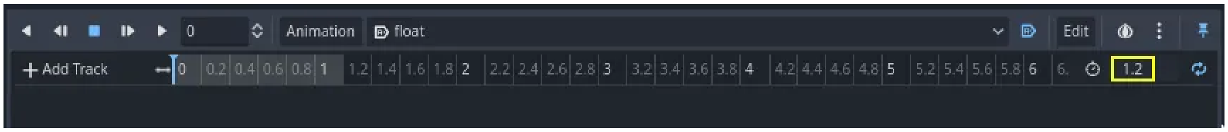
To do so, you can click the autoplay button () in the animation toolbar and the looping arrows, respectively.


0
Animation
float
Edit
+Add Track
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
1
Snap:
0.1
Seconds
Output
Debugger
Audio
Animation
3.2.2.stable

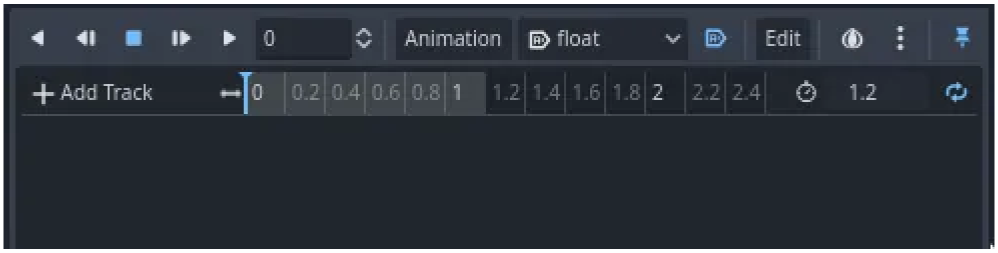
You can also pin the animation editor by clicking the pin icon in the top-right. This prevents it from folding when you click on the viewport and deselect the nodes.



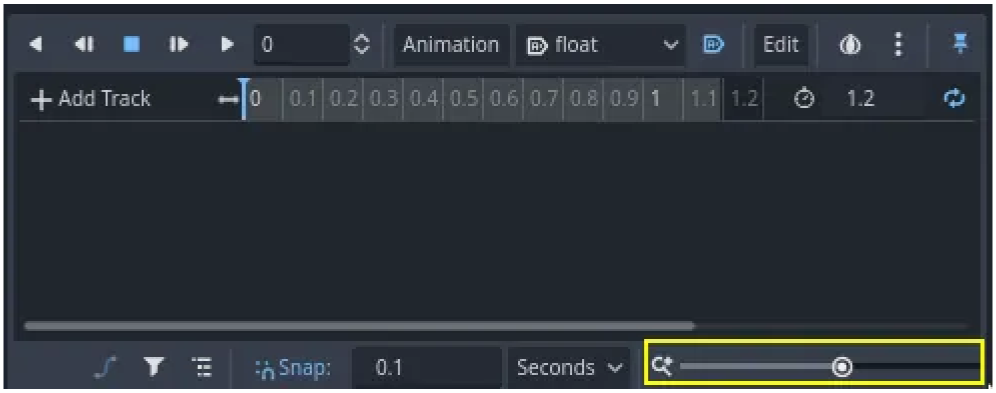
Set the animation duration to `1.2` seconds in the top-right of the dock.


0
Animation
float
Edit
Add Track
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
2.8
3
3.2
3.4
3.6
3.8
4
4.2
4.4
4.6
4.8
5
5.2
5.4
5.6
5.8
6
6.
1.2

You should see the gray ribbon widen a bit. It shows you the start and end of your animation and the vertical blue line is your time cursor.


0
Animation
float
Edit
Add Track
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2 2.2 2.4
1.2

You can click and drag the slider in the bottom-right to zoom in and out of the timeline.

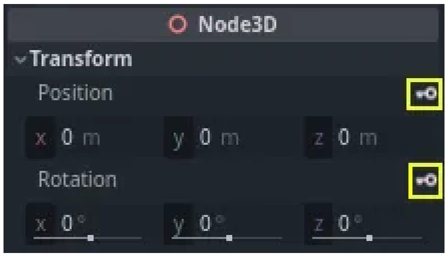

0
Animation
float
Edit
Add Track
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2
1.2
Snap:
0.1
Seconds

The float animation

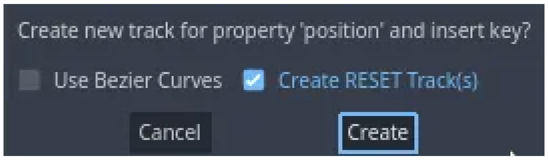
With the animation player node, you can animate most properties on as many nodes as you need. Notice the key icon next to properties in the *Inspector*. You can click any of them to create a keyframe, a time and value pair for the corresponding property. The keyframe gets inserted where your time cursor is in the timeline.

Let's insert our first keys. Here, we will animate both the position and the rotation of the `Character` node.

Select the `Character` and in the *Inspector* expand the *Transform* section. Click the key icon next to *Position*, and *Rotation*.


Node3D
Transform

Position
x 0 m y 0 m z 0 m
Rotation
x 0° y 0° z 0°



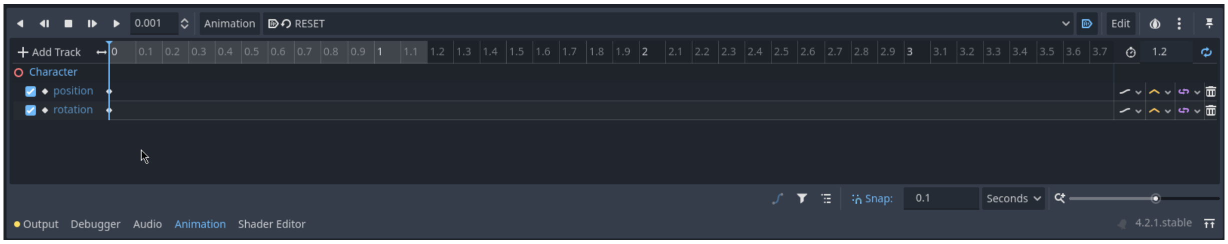
Create new track for property 'position' and insert key?

☐ Use Bezier Curves ☑ Create RESET Track(s)

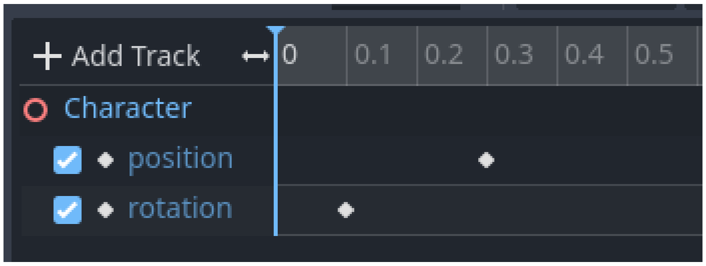
Cancel
Create

For this tutorial, just create RESET Track(s) which is the default choice

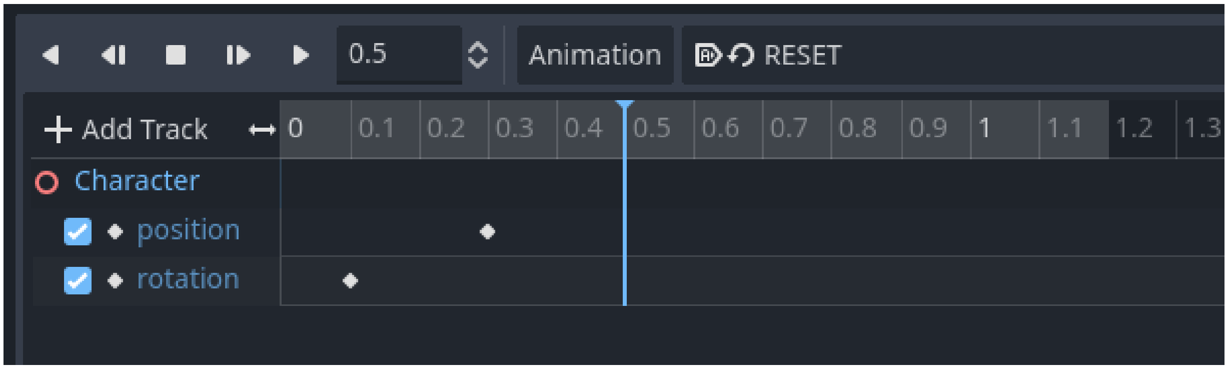
Two tracks appear in the editor with a diamond icon representing each keyframe.


0.001
Animation
RESET
Edit
Add Track
Character
position
rotation
1.2
Snap: 0.1
Seconds
Output
Debugger
Audio
Animation
Shader Editor
4.2.1.stable

You can click and drag on the diamonds to move them in time. Move the position key to `0.3` seconds and the rotation key to `0.1` seconds.

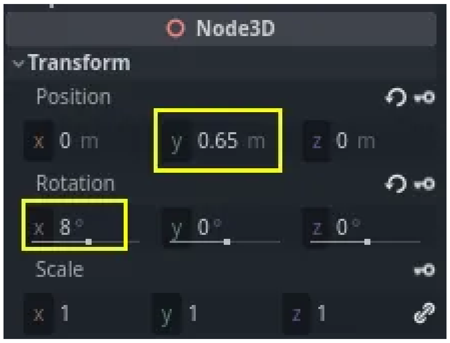

Add Track
0
0.1
0.2
0.3
0.4
0.5
Character
position
rotation

Move the time cursor to `0.5` seconds by clicking and dragging on the gray timeline.

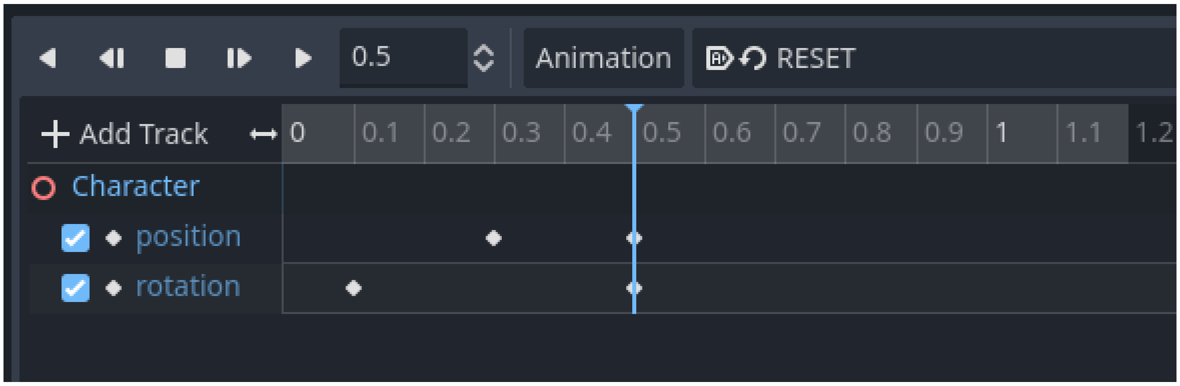

0.5
Animation
RESET
Add Track
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2 1.3
Character
position
rotation

In the *Inspector*, set the *Position*'s *Y* axis to `0.65` meters and the *Rotation*' *X* axis to `8`.

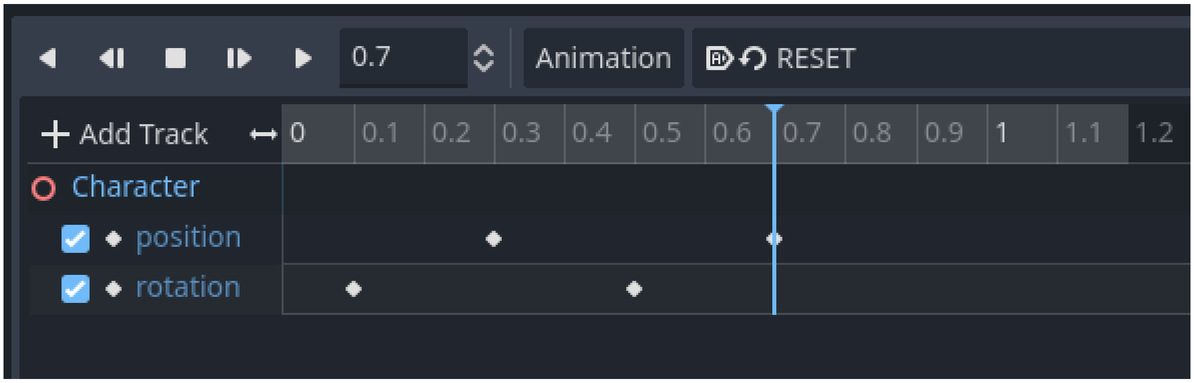
If you don't see the properties in the *Inspector* panel, first click on the `Character` node again in the *Scene* dock.


Node3D
Transform
Position
x 0 m
y 0.65 m
z 0 m
Rotation
x 8°
y 0°
z 0°
Scale
x 1
y 1
z 1

Create a keyframe for both properties


0.5
Animation
RESET
Add Track
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2
Character
position
rotation

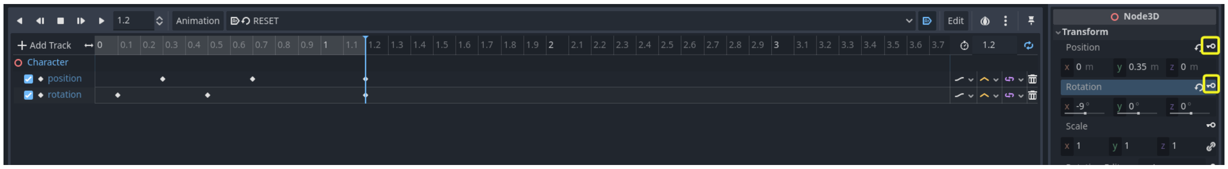
Now, move the position keyframe to `0.7` seconds by dragging it on the timeline.


0.7
Animation
RESET
Add Track
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
1.2
Character
position
rotation

Note

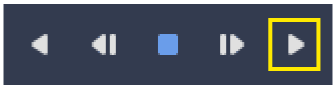
A lecture on the principles of animation is beyond the scope of this tutorial. Just note that you don't want to time and space everything evenly. Instead, animators play with timing and spacing, two core animation principles. You want to offset and contrast in your character's motion to make them feel alive.

Move the time cursor to the end of the animation, at `1.2` seconds. Set the Y position to about `0.35` and the X rotation to `-9` degrees. Once again, create a key for both properties.


1.2
Animation
RESET
Edit
Add Track
Character
position
rotation
Node3D
Transform
Position
Rotation
Scale

Position: x 0 m, y 0.35 m, z 0 m
Rotation: x -9°, y 0°, z 0°
Scale: x 1, y 1, z 1

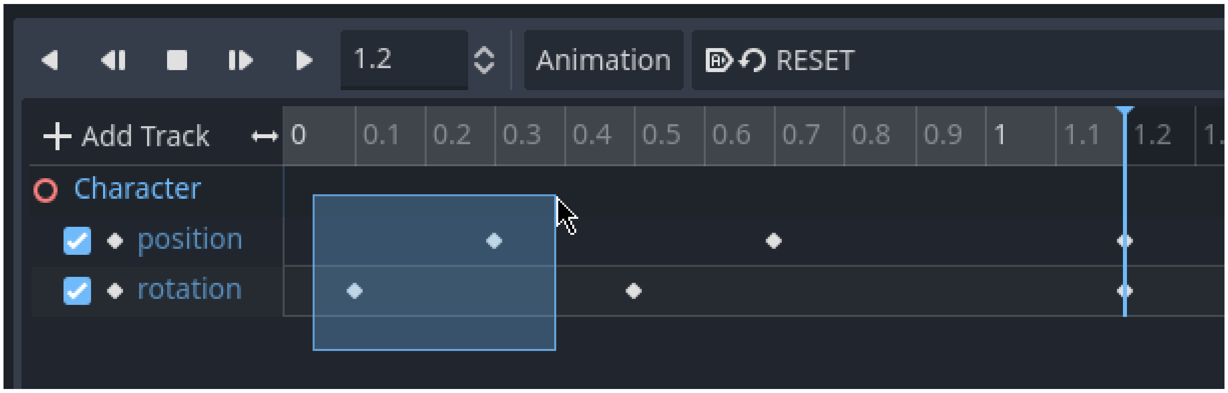
You can preview the result by clicking the play button or pressing `Shift + D`. Click the stop button or press `S` to stop playback.



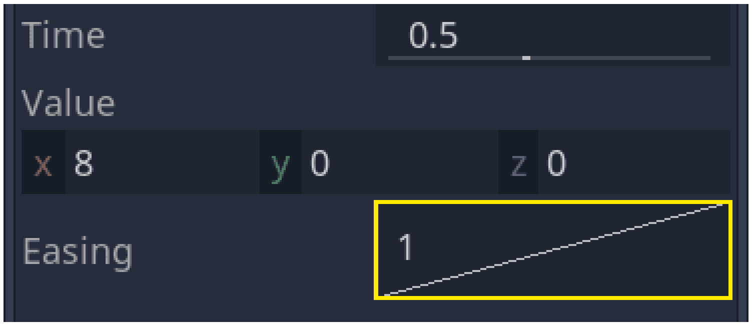
You can see that the engine interpolates between your keyframes to produce a continuous animation. At the moment, though, the motion feels very robotic. This is because the default interpolation is linear, causing constant transitions, unlike how living things move in the real world.

We can control the transition between keyframes using easing curves.

Click and drag around the first two keys in the timeline to box select them.


1.2
Animation
RESET
Add Track
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
1.2
Character
position
rotation

You can edit the properties of both keys simultaneously in the *Inspector*, where you can see an *Easing* property.



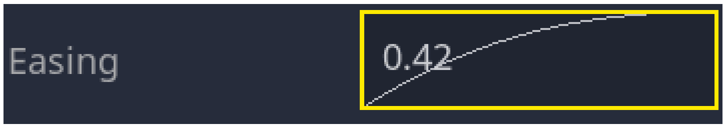
Time 0.5

Value

x 8 y 0 z 0

Easing 1

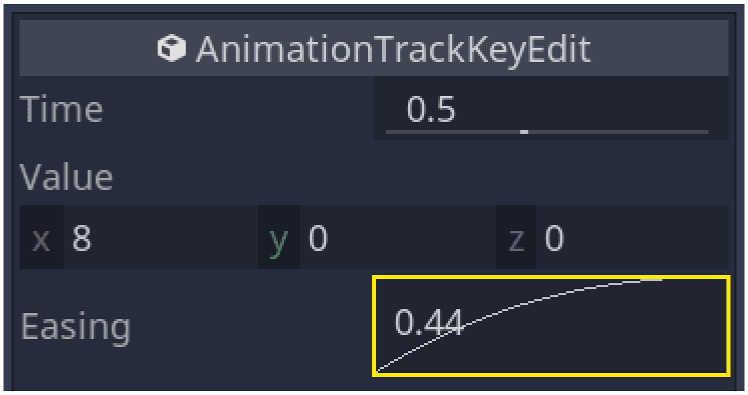
Click and drag on the curve, pulling it towards the left. This will make it ease-out, that is to say, transition fast initially and slow down as the time cursor reaches the next keyframe.



Easing 0.42

Play the animation again to see the difference. The first half should already feel a bit bouncier.

Apply an ease-out to the second keyframe in the rotation track.



AnimationTrackKeyEdit

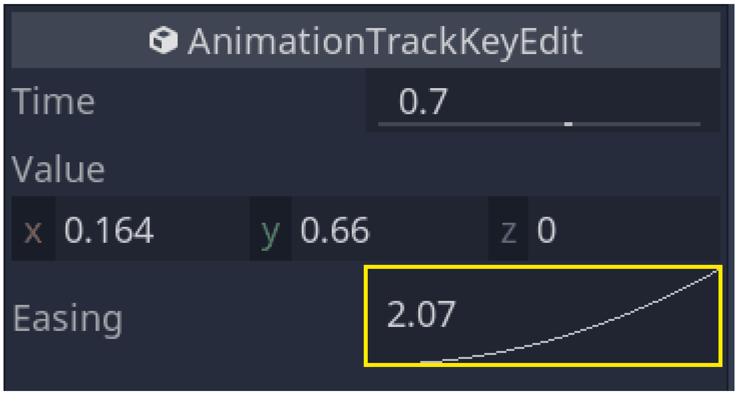
Time 0.5

Value

x 8 y 0 z 0

Easing 0.44

Do the opposite for the second position keyframe, dragging it to the right.



AnimationTrackKeyEdit

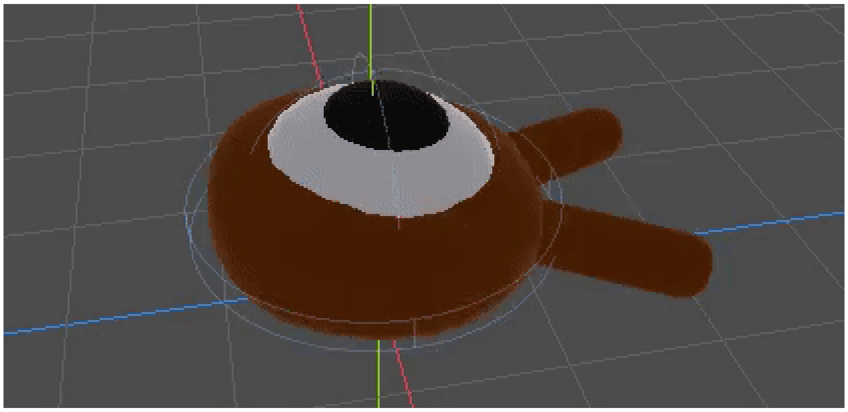
Time 0.7

Value

x 0.164 y 0.66 z 0

Easing 2.07

Your animation should look something like this.



Note

Animations update the properties of the animated nodes every frame, overriding initial values. If we directly animated the *Player* node, it would prevent us from moving it in code. This is where the *Pivot* node comes in handy: even though we animated the *Character*, we can still move and rotate the *Pivot* and layer changes on top of the animation in a script.

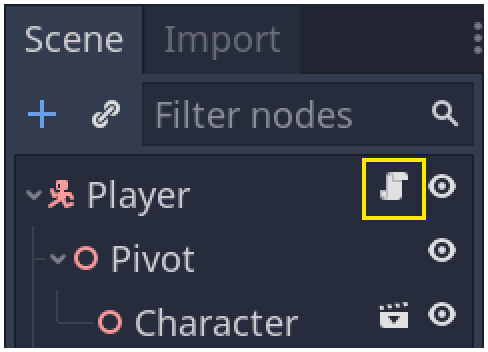
If you play the game, the player's creature will now float!

If the creature is a little too close to the floor, you can move the `Pivot` up to offset it.

Controlling the animation in code

We can use code to control the animation playback based on the player's input. Let's change the animation speed when the character is moving.

Open the `Player`'s script by clicking the script icon next to it.



In `_physics_process()`, after the line where we check the `direction` vector, add the following code.

GDScript

```
func _physics_process(delta):
    #...
    if direction != Vector3.ZERO:
        #...
        $AnimationPlayer.speed_scale = 4
    else:
        $AnimationPlayer.speed_scale = 1
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...
    if (direction != Vector3.Zero)
    {
        // ...
        GetNode<AnimationPlayer>("AnimationPlayer").SpeedScale
= 4;
    }
    else
    {
        GetNode<AnimationPlayer>("AnimationPlayer").SpeedScale
= 1;
    }
}
```

This code makes it so when the player moves, we multiply the playback speed by `4`. When they stop, we reset it to normal.

We mentioned that the `Pivot` could layer transforms on top of the animation. We can make the character arc when jumping using the following line of code. Add it at the end of `_physics_process()`.

GDScript

```
func _physics_process(delta):
    #...
    $Pivot.rotation.x = PI / 6 * velocity.y / jump_impulse
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...
    var pivot = GetNode<Node3D>("Pivot");
    pivot.Rotation = new Vector3(Mathf.Pi / 6.0f * Velocity.Y / JumpImpulse, pivot.Rotation.Y, pivot.Rotation.Z);
}
```

Animating the mobs

Here's another nice trick with animations in Godot: as long as you use a similar node structure, you can copy them to different scenes.

For example, both the `Mob` and the `Player` scenes have a `Pivot` and a `Character` node, so we can reuse animations between them.

Open the *Player* scene, select the AnimationPlayer node and then click on **Animation > Manage Animations...**. Click the *Copy animation to clipboard* button (two small squares) alongside the *float* animation. Click OK to close the window.

Then open `mob.tscn`, create an AnimationPlayer child node and select it. Click **Animation > Manage Animations**, then **Add Library**. You should see the message "Global library will be created." Leave the text field blank and click OK. Click the *Paste* icon (clipboard) and it should appear in the window. Click OK to close the window.

Next, make sure that the autoplay button (A) and the looping arrows (Animation looping) are also turned on in the animation editor in the bottom panel. That's it; all monsters will now play the float animation.

We can change the playback speed based on the creature's `random_speed`. Open the *Mob*'s script and at the end of the `initialize()` function, add the following line.

GDScript

```
func initialize(start_position, player_position):
    #...
    $AnimationPlayer.speed_scale = random_speed / min_speed
```

C#

```
public void Initialize(Vector3 startPosition, Vector3
playerPosition)
{
    // ...
    GetNode<AnimationPlayer>("AnimationPlayer").SpeedScale =
randomSpeed / MinSpeed;
}
```

And with that, you finished coding your first complete 3D game.

Congratulations!

In the next part, we'll quickly recap what you learned and give you some links to keep learning more. But for now, here are the complete `player.gd` and `mob.gd` so you can check your code against them.

Here's the *Player* script.

GDScript

```
extends CharacterBody3D

signal hit

# How fast the player moves in meters per second.
@export var speed = 14
# The downward acceleration while in the air, in meters per
second squared.
@export var fall_acceleration = 75
# Vertical impulse applied to the character upon jumping in
meters per second.
@export var jump_impulse = 20
```

```
# Vertical impulse applied to the character upon bouncing over
a mob
# in meters per second.
@export var bounce_impulse = 16

var target_velocity = Vector3.ZERO


func _physics_process(delta):
    # We create a local variable to store the input direction
    var direction = Vector3.ZERO

    # We check for each move input and update the direction
accordingly
    if Input.is_action_pressed("move_right"):
        direction.x = direction.x + 1
    if Input.is_action_pressed("move_left"):
        direction.x = direction.x - 1
    if Input.is_action_pressed("move_back"):
        # Notice how we are working with the vector's x and z
axes.
        # In 3D, the XZ plane is the ground plane.
        direction.z = direction.z + 1
    if Input.is_action_pressed("move_forward"):
        direction.z = direction.z - 1

    # Prevent diagonal movement being very fast
    if direction != Vector3.ZERO:
        direction = direction.normalized()
        $Pivot.look_at(position + direction,Vector3.UP)
        $AnimationPlayer.speed_scale = 4
    else:
        $AnimationPlayer.speed_scale = 1

    # Ground Velocity
    target_velocity.x = direction.x * speed
    target_velocity.z = direction.z * speed

    # Vertical Velocity
    if not is_on_floor(): # If in the air, fall towards the
floor
        target_velocity.y = target_velocity.y -
(fall_acceleration * delta)
```

```
    # Jumping.
    if is_on_floor() and Input.is_action_just_pressed("jump"):
        target_velocity.y = jump_impulse

    # Iterate through all collisions that occurred this frame
    # in C this would be for(int i = 0; i < collisions.Count; i++)
    for index in range(get_slide_collision_count()):
        # We get one of the collisions with the player
        var collision = get_slide_collision(index)

        # If the collision is with ground
        if collision.get_collider() == null:
            continue

        # If the collider is with a mob
        if collision.get_collider().is_in_group("mob"):
            var mob = collision.get_collider()
            # we check that we are hitting it from above.
            if Vector3.UP.dot(collision.get_normal()) > 0.1:
                # If so, we squash it and bounce.
                mob.squash()
                target_velocity.y = bounce_impulse
                # Prevent further duplicate calls.
                break

    # Moving the Character
    velocity = target_velocity
    move_and_slide()

    $Pivot.rotation.x = PI / 6 * velocity.y / jump_impulse

# And this function at the bottom.
func die():
    hit.emit()
    queue_free()

func _on_mob_detector_body_entered(body):
    die()
```

C#

```
using Godot;

public partial class Player : CharacterBody3D
{
    // Emitted when the player was hit by a mob.
    [Signal]
    public delegate void HitEventHandler();

    // How fast the player moves in meters per second.
    [Export]
    public int Speed { get; set; } = 14;
    // The downward acceleration when in the air, in meters per
second squared.
    [Export]
    public int FallAcceleration { get; set; } = 75;
    // Vertical impulse applied to the character upon jumping
in meters per second.
    [Export]
    public int JumpImpulse { get; set; } = 20;
    // Vertical impulse applied to the character upon bouncing
over a mob in meters per second.
    [Export]
    public int BounceImpulse { get; set; } = 16;

    private Vector3 _targetVelocity = Vector3.Zero;

    public override void _PhysicsProcess(double delta)
    {
        // We create a local variable to store the input
direction.
        var direction = Vector3.Zero;

        // We check for each move input and update the
direction accordingly.
        if (Input.IsActionPressed("move_right"))
        {
            direction.X += 1.0f;
        }
        if (Input.IsActionPressed("move_left"))
        {
            direction.X -= 1.0f;
        }
        if (Input.IsActionPressed("move_back"))
        {
            // Notice how we are working with the vector's X
and Z axes.
            // In 3D, the XZ plane is the ground plane.
            direction.Z += 1.0f;
        }
        if (Input.IsActionPressed("move_forward"))
        {
            direction.Z -= 1.0f;
        }

        // Prevent diagonal movement being very fast.
        if (direction != Vector3.Zero)
        {
            direction = direction.Normalized();
            GetNode<Node3D>("Pivot").LookAt(Position +
direction, Vector3.Up);
            GetNode<AnimationPlayer>
("AnimationPlayer").PlaybackSpeed = 4;
        }
        else
        {
            GetNode<AnimationPlayer>
("AnimationPlayer").PlaybackSpeed = 1;
        }

        // Ground velocity
        _targetVelocity.X = direction.X * Speed;
        _targetVelocity.Z = direction.Z * Speed;

        // Vertical velocity
        if (!IsOnFloor())
        {
            _targetVelocity.Y -= FallAcceleration *
(float)delta;
        }

        // Jumping.
        if (IsOnFloor() && Input.IsActionJustPressed("jump"))
        {
            _targetVelocity.Y += JumpImpulse;
        }
        // Iterate through all collisions that occurred this
frame.
        for (int index = 0; index < GetSlideCollisionCount();
index++)
        {
            // We get one of the collisions with the player.
            KinematicCollision3D collision =
GetSlideCollision(index);

            // If the collision is with a mob.
            if (collision.GetCollider() is Mob mob)
            {
                // We check that we are hitting it from above.
                if (Vector3.Up.Dot(collision.GetNormal()) >
0.1f)
                {
                    // If so, we squash it and bounce.
                    mob.Squash();
                    _targetVelocity.Y = BounceImpulse;
                    // Prevent further duplicate calls.
                    break;
                }
            }
        }

        // Moving the character
        Velocity = _targetVelocity;
        MoveAndSlide();

        var pivot = GetNode<Node3D>("Pivot");
        pivot.Rotation = new Vector3(Mathf.Pi / 6.0f *
Velocity.Y / JumpImpulse, pivot.Rotation.Y, pivot.Rotation.Z);
    }

    private void Die()
    {
        EmitSignal(SignalName.Hit);
        QueueFree();
    }

    private void OnMobDetectorBodyEntered(Node body)
    {
        Die();
    }
}
```

And the *Mob*'s script.

GDScript

```
extends CharacterBody3D

# Minimum speed of the mob in meters per second.
@export var min_speed = 10
# Maximum speed of the mob in meters per second.
@export var max_speed = 18

# Emitted when the player jumped on the mob
signal squashed

func _physics_process(_delta):
    move_and_slide()

# This function will be called from the Main scene.
func initialize(start_position, player_position):
    # We position the mob by placing it at start_position
    # and rotate it towards player_position, so it looks at the player.
    look_at_from_position(start_position, player_position, Vector3.UP)
    # Rotate this mob randomly within range of -45 and +45 degrees,
    # so that it doesn't move directly towards the player.
    rotate_y(randf_range(-PI / 4, PI / 4))

    # We calculate a random speed (integer)
    var random_speed = randi_range(min_speed, max_speed)
    # We calculate a forward velocity that represents the speed.
    velocity = Vector3.FORWARD * random_speed
    # We then rotate the velocity vector based on the mob's Y rotation
    # in order to move in the direction the mob is looking.
    velocity = velocity.rotated(Vector3.UP, rotation.y)
    $AnimationPlayer.speed_scale = random_speed / min_speed

func _on_visible_on_screen_notifier_3d_screen_exited():
    queue_free()

func squash():
    squashed.emit()
    queue_free() # Destroy this node
```

C#

```
using Godot;

public partial class Mob : CharacterBody3D
{
    // Emitted when the played jumped on the mob.
    [Signal]
    public delegate void SquashedEventHandler();

    // Minimum speed of the mob in meters per second
    [Export]
    public int MinSpeed { get; set; } = 10;
    // Maximum speed of the mob in meters per second
    [Export]
    public int MaxSpeed { get; set; } = 18;

    public override void _PhysicsProcess(double delta)
    {
        MoveAndSlide();
    }

    // This function will be called from the Main scene.
    public void Initialize(Vector3 startPosition, Vector3
playerPosition)
    {
        // We position the mob by placing it at startPosition
        // and rotate it towards playerPosition, so it looks at
the player.
        LookAtFromPosition(startPosition, playerPosition,
Vector3.Up);
        // Rotate this mob randomly within range of -45 and +45
degrees,
        // so that it doesn't move directly towards the player.
        RotateY((float)GD.RandRange(-Mathf.Pi / 4.0, Mathf.Pi /
4.0));

        // We calculate a random speed (integer).
        int randomSpeed = GD.RandRange(MinSpeed, MaxSpeed);
        // We calculate a forward velocity that represents the
speed.
        Velocity = Vector3.Forward * randomSpeed;
        // We then rotate the velocity vector based on the
mob's Y rotation
        // in order to move in the direction the mob is
looking.
        Velocity = Velocity.Rotated(Vector3.Up, Rotation.Y);

        GetNode<AnimationPlayer>("AnimationPlayer").SpeedScale
= randomSpeed / MinSpeed;
    }

    public void Squash()
    {
        EmitSignal(SignalName.Squashed);
        QueueFree(); // Destroy this node
    }

    private void OnVisibilityNotifierScreenExited()
    {
        QueueFree();
    }
}
```

Going further

You can pat yourself on the back for having completed your first 3D game with Godot.

In this series, we went over a wide range of techniques and editor features. Hopefully, you've witnessed how intuitive Godot's scene system can be and learned a few tricks you can apply in your projects.

But we just scratched the surface: Godot has a lot more in store for you to save time creating games. And you can learn all that by browsing the documentation.

Where should you begin? Below, you'll find a few pages to start exploring and build upon what you've learned so far.

But before that, here's a link to download a completed version of the project: https://github.com/godotengine/godot-demo-projects/releases.

Exploring the manual

The manual is your ally whenever you have a doubt or you're curious about a feature. It does not contain tutorials about specific game genres or mechanics. Instead, it explains how Godot works in general. In it, you will find information about 2D, 3D, physics, rendering and performance, and much more.

Here are the sections we recommend you to explore next:

1. Read the Scripting section to learn essential programming features you'll use in every project.

2. The 3D and Physics sections will teach you more about 3D game creation in the engine.
3. Inputs is another important one for any game project.

You can start with these or, if you prefer, look at the sidebar menu on the left and pick your options.

We hope you enjoyed this tutorial series, and we're looking forward to seeing what you achieve using Godot.

Best practices

- Introduction
- Applying object-oriented principles in Godot
- Scene organization
- When to use scenes versus scripts
- Autoloads versus regular nodes
- When and how to avoid using nodes for everything
- Godot interfaces
- Godot notifications
- Data preferences
- Logic preferences
- Project organization
- Version control systems

Introduction

This series is a collection of best practices to help you work efficiently with Godot.

Godot allows for a great amount of flexibility in how you structure a project's codebase and break it down into scenes. Each approach has its pros and cons, and they can be hard to weigh until you've worked with the engine for long enough.

There are always many ways to structure your code and solve specific programming problems. It would be impossible to cover them all here.

That is why each article starts from a real-world problem. We will break down each problem in fundamental questions, suggest solutions, analyze the pros and cons of each option, and highlight the best course of action for the problem at hand.

You should start by reading Applying object-oriented principles in Godot. It explains how Godot's nodes and scenes relate to classes and objects in other Object-Oriented programming languages. It will help you make sense of the rest of the series.

Note

The best practices in Godot rely on Object-Oriented design principles. We use tools like the single responsibility [https://en.wikipedia.org/wiki/Single_responsibility_principle] principle and encapsulation [https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)].

Applying object-oriented principles in Godot

The engine offers two main ways to create reusable objects: scripts and scenes. Neither of these technically define classes under the hood.

Still, many best practices using Godot involve applying object-oriented programming principles to the scripts and scenes that compose your game. That is why it's useful to understand how we can think of them as classes.

This guide briefly explains how scripts and scenes work in the engine's core to help you understand how they work under the hood.

How scripts work in the engine

The engine provides built-in classes like Node. You can extend those to create derived types using a script.

These scripts are not technically classes. Instead, they are resources that tell the engine a sequence of initializations to perform on one of the engine's built-in classes.

Godot's internal classes have methods that register a class's data with a ClassDB. This database provides runtime access to class information. `ClassDB` contains information about classes like:

- Properties.
- Methods.
- Constants.

- Signals.

This `ClassDB` is what objects check against when performing an operation like accessing a property or calling a method. It checks the database's records and the object's base types' records to see if the object supports the operation.

Attaching a Script to your object extends the methods, properties, and signals available from the `ClassDB`.

Note

Even scripts that don't use the `extends` keyword implicitly inherit from the engine's base RefCounted class. As a result, you can instantiate scripts without the `extends` keyword from code. Since they extend `RefCounted` though, you cannot attach them to a Node.

Scenes

The behavior of scenes has many similarities to classes, so it can make sense to think of a scene as a class. Scenes are reusable, instantiable, and inheritable groups of nodes. Creating a scene is similar to having a script that creates nodes and adds them as children using `add_child()`.

We often pair a scene with a scripted root node that makes use of the scene's nodes. As such, the script extends the scene by adding behavior through imperative code.

The content of a scene helps to define:

- What nodes are available to the script.
- How they are organized.
- How they are initialized.

- What signal connections they have with each other.

Why is any of this important to scene organization? Because instances of scenes *are* objects. As a result, many object-oriented principles that apply to written code also apply to scenes: single responsibility, encapsulation, and others.

The scene is *always an extension of the script attached to its root node*, so you can interpret it as part of a class.

Most of the techniques explained in this best practices series build on this point.

Scene organization

This article covers topics related to the effective organization of scene content. Which nodes should you use? Where should you place them? How should they interact?

How to build relationships effectively

When Godot users begin crafting their own scenes, they often run into the following problem:

They create their first scene and fill it with content only to eventually end up saving branches of their scene into separate scenes as the nagging feeling that they should split things up starts to accumulate. However, they then notice that the hard references they were able to rely on before are no longer possible. Re-using the scene in multiple places creates issues because the node paths do not find their targets and signal connections established in the editor break.

To fix these problems, you must instantiate the sub-scenes without them requiring details about their environment. You need to be able to trust that the sub-scene will create itself without being picky about how it's used.

One of the biggest things to consider in OOP is maintaining focused, singular-purpose classes with loose coupling [https://en.wikipedia.org/wiki/Loose_coupling] to other parts of the codebase. This keeps the size of objects small (for maintainability) and improves their reusability.

These OOP best practices have *several* implications for best practices in scene structure and script usage.

If at all possible, you should design scenes to have no dependencies. That is, you should create scenes that keep everything they need within themselves.

If a scene must interact with an external context, experienced developers recommend the use of [Dependency Injection](https://en.wikipedia.org/wiki/Dependency_injection) [https://en.wikipedia.org/wiki/Dependency_injection]. This technique involves having a high-level API provide the dependencies of the low-level API. Why do this? Because classes which rely on their external environment can inadvertently trigger bugs and unexpected behavior.

To do this, you must expose data and then rely on a parent context to initialize it:

1. Connect to a signal. Extremely safe, but should be used only to "respond" to behavior, not start it. By convention, signal names are usually past-tense verbs like "entered", "skill_activated", or "item_collected".

   GDScript

   ```
   # Parent
   $Child.signal_name.connect(method_on_the_object)

   # Child
   signal_name.emit() # Triggers parent-defined behavior.
   ```

   C#

   ```
   // Parent
   GetNode("Child").Connect("SignalName",
   Callable.From(ObjectWithMethod.MethodOnTheObject));

   // Child
   ```

```
EmitSignal("SignalName"); // Triggers parent-defined
behavior.
```

2. Call a method. Used to start behavior.

   GDScript

   ```
   # Parent
   $Child.method_name = "do"

   # Child, assuming it has String property 'method_name' and
   method 'do'.
   call(method_name) # Call parent-defined method (which child
   must own).
   ```

   C#

   ```
   // Parent
   GetNode("Child").Set("MethodName", "Do");

   // Child
   Call(MethodName); // Call parent-defined method (which
   child must own).
   ```

3. Initialize a Callable property. Safer than a method as ownership of the method is unnecessary. Used to start behavior.

   GDScript

   ```
   # Parent
   $Child.func_property =
   object_with_method.method_on_the_object

   # Child
   func_property.call() # Call parent-defined method (can come
   from anywhere).
   ```

   C#

```
// Parent
GetNode("Child").Set("FuncProperty",
Callable.From(ObjectWithMethod.MethodOnTheObject));

// Child
FuncProperty.Call(); // Call parent-defined method (can
come from anywhere).
```

4. Initialize a Node or other Object reference.

GDScript

```
# Parent
$Child.target = self

# Child
print(target) # Use parent-defined node.
```

C#

```
// Parent
GetNode("Child").Set("Target", this);

// Child
GD.Print(Target); // Use parent-defined node.
```

5. Initialize a NodePath.

GDScript

```
# Parent
$Child.target_path = ".."

# Child
get_node(target_path) # Use parent-defined NodePath.
```

C#

```
// Parent
GetNode("Child").Set("TargetPath", NodePath(".."));
```

```
// Child
GetNode(TargetPath); // Use parent-defined NodePath.
```

These options hide the points of access from the child node. This in turn keeps the child **loosely coupled** to its environment. You can reuse it in another context without any extra changes to its API.

Note

Although the examples above illustrate parent-child relationships, the same principles apply towards all object relations. Nodes which are siblings should only be aware of their own hierarchies while an ancestor mediates their communications and references.

GDScript

```
# Parent
$Left.target = $Right.get_node("Receiver")

# Left
var target: Node
func execute():
    # Do something with 'target'.

# Right
func _init():
    var receiver = Receiver.new()
    add_child(receiver)
```

C#

```
// Parent
GetNode<Left>("Left").Target = GetNode("Right/Receiver");

public partial class Left : Node
{
    public Node Target = null;

    public void Execute()
    {
        // Do something with 'Target'.
    }
}

public partial class Right : Node
{
    public Node Receiver = null;

    public Right()
    {
        Receiver = ResourceLoader.Load<Script>
("Receiver.cs").New();
        AddChild(Receiver);
    }
}
```

The same principles also apply to non-Node objects that maintain dependencies on other objects. Whichever object owns the other objects should manage the relationships between them.

Warning

You should favor keeping data in-house (internal to a scene), though, as placing a dependency on an external context, even a loosely coupled one, still means that the node will expect something in its environment to be true. The project's design philosophies should prevent this from happening. If not, the code's inherent liabilities will force developers to use documentation to keep track of object relations on a microscopic scale; this is otherwise known as development hell. Writing code that relies on external documentation to use it safely is error-prone by default.

To avoid creating and maintaining such documentation, you convert the dependent node ("child" above) into a tool script that implements `_get_configuration_warnings()`. Returning a non-empty PackedStringArray from it will make the Scene dock generate a warning icon with the string(s) as a tooltip by the node.

This is the same icon that appears for nodes such as the Area2D node when it has no child CollisionShape2D nodes defined. The editor then self-documents the scene through the script code. No content duplication via documentation is necessary.

A GUI like this can better inform project users of critical information about a Node. Does it have external dependencies? Have those dependencies been satisfied? Other programmers, and especially designers and writers, will need clear instructions in the messages telling them what to do to configure it.

So, why does all this complex switcheroo work? Well, because scenes operate best when they operate alone. If unable to work alone, then working with others anonymously (with minimal hard dependencies, i.e. loose coupling) is the next best thing. Inevitably, changes may need to be made to a class, and if these changes cause it to interact with other scenes in unforeseen ways, then things will start to break down. The whole point of all this indirection is to avoid ending up in a situation where changing one class results in adversely affecting other classes dependent on it.

Scripts and scenes, as extensions of engine classes, should abide by *all* OOP principles. Examples include...

- SOLID [https://en.wikipedia.org/wiki/SOLID]
- DRY [https://en.wikipedia.org/wiki/Don%27t_repeat_yourself]
- KISS [https://en.wikipedia.org/wiki/KISS_principle]
- YAGNI [https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it]

Choosing a node tree structure

You might start to work on a game but get overwhelmed by the vast possibilities before you. You might know what you want to do, what systems you want to have, but *where* do you put them all? How you go about making your game is always up to you. You can construct node trees in countless ways. If you are unsure, this guide can give you a sample of a decent structure to start with.

A game should always have an "entry point"; somewhere you can definitively track where things begin so that you can follow the logic as it continues elsewhere. It also serves as a bird's eye view of all other data and logic in the program. For traditional applications, this is normally a "main" function. In Godot, it's a Main node.

- Node "Main" (main.gd)

The `main.gd` script will serve as the primary controller of your game.

Then you have an in-game "World" (a 2D or 3D one). This can be a child of Main. In addition, you will need a primary GUI for your game that manages the various menus and widgets the project needs.

- Node "Main" (main.gd)
  - Node2D/Node3D "World" (game_world.gd)
  - Control "GUI" (gui.gd)

When changing levels, you can then swap out the children of the "World" node. Changing scenes manually gives you full control over how your game world transitions.

The next step is to consider what gameplay systems your project requires. If you have a system that...

1. tracks all of its data internally
2. should be globally accessible
3. should exist in isolation

... then you should create an autoload 'singleton' node.

Note

For smaller games, a simpler alternative with less control would be to have a "Game" singleton that simply calls the SceneTree.change_scene_to_file() method to swap out the main scene's content. This structure more or less keeps the "World" as the main game node.

Any GUI would also need to be either a singleton, a transitory part of the "World", or manually added as a direct child of the root. Otherwise, the GUI nodes would also delete themselves during scene transitions.

If you have systems that modify other systems' data, you should define those as their own scripts or scenes, rather than autoloads. For more information, see Autoloads versus regular nodes.

Each subsystem within your game should have its own section within the SceneTree. You should use parent-child relationships only in cases where nodes are effectively elements of their parents. Does removing the parent reasonably mean that the children should also be removed? If not, then it should have its own place in the hierarchy as a sibling or some other relation.

Note

In some cases, you need these separated nodes to *also* position themselves relative to each other. You can use the RemoteTransform / RemoteTransform2D nodes for this purpose. They will allow a target node to conditionally inherit selected transform elements from the Remote* node. To assign the `target` NodePath, use one of the following:

1. A reliable third party, likely a parent node, to mediate the assignment.
2. A group, to pull a reference to the desired node (assuming there will only ever be one of the targets).

When you should do this is subjective. The dilemma arises when you must micro-manage when a node must move around the SceneTree to preserve itself. For example...

- Add a "player" node to a "room".
- Need to change rooms, so you must delete the current room.
- Before the room can be deleted, you must preserve and/or move the player.

  If memory is not a concern, you can...

  - Create the new room.
  - Move the player to the new room.
  - Delete the old room.

  If memory is a concern, instead you will need to...

  - Move the player somewhere else in the tree.
  - Delete the room.
  - Instantiate and add the new room.
  - Re-add the player to the new room.

The issue is that the player here is a "special case" where the developers must *know* that they need to handle the player this way for the project. The only way to reliably share this information as a team is to *document* it. Keeping implementation details in documentation is dangerous. It's a maintenance burden, strains code readability, and unnecessarily bloats the intellectual content of a project.

In a more complex game with larger assets, it can be a better idea to keep the player somewhere else in the SceneTree entirely. This results in:

1. More consistency.

2. No "special cases" that must be documented and maintained somewhere.
3. No opportunity for errors to occur because these details are not accounted for.

In contrast, if you ever need a child node that does *not* inherit the transform of its parent, you have the following options:

1. The **declarative** solution: place a Node in between them. Since it doesn't have a transform, they won't pass this information to its children.
2. The **imperative** solution: Use the `top_level` property for the CanvasItem or Node3D node. This will make the node ignore its inherited transform.

Note

If building a networked game, keep in mind which nodes and gameplay systems are relevant to all players versus those just pertinent to the authoritative server. For example, users do not all need to have a copy of every players' "PlayerController" logic - they only need their own. Keeping them in a separate branch from the "world" can help simplify the management of game connections and the like.

The key to scene organization is to consider the SceneTree in relational terms rather than spatial terms. Are the nodes dependent on their parent's existence? If not, then they can thrive all by themselves somewhere else. If they are dependent, then it stands to reason that they should be children of that parent (and likely part of that parent's scene if they aren't already).

Does this mean nodes themselves are components? Not at all. Godot's node trees form an aggregation relationship, not one of composition. But while you still have the flexibility to move nodes around, it is still best when such moves are unnecessary by default.

When to use scenes versus scripts

We've already covered how scenes and scripts are different. Scripts define an engine class extension with imperative code, scenes with declarative code.

Each system's capabilities are different as a result. Scenes can define how an extended class initializes, but not what its behavior actually is. Scenes are often used in conjunction with a script, the scene declaring a composition of nodes, and the script adding behaviour with imperative code.

Anonymous types

It *is* possible to completely define a scenes' contents using a script alone. This is, in essence, what the Godot Editor does, only in the C++ constructor of its objects.

But, choosing which one to use can be a dilemma. Creating script instances is identical to creating in-engine classes whereas handling scenes requires a change in API:

GDScript

```
const MyNode = preload("my_node.gd")
const MyScene = preload("my_scene.tscn")
var node = Node.new()
var my_node = MyNode.new() # Same method call.
var my_scene = MyScene.instantiate() # Different method call.
var my_inherited_scene =
MyScene.instantiate(PackedScene.GEN_EDIT_STATE_MAIN) # Create
scene inheriting from MyScene.
```

C#

```
using Godot;

public partial class Game : Node
{
    public static CSharpScript MyNode { get; } =
        GD.Load<CSharpScript>("res://Path/To/MyNode.cs");
    public static PackedScene MyScene { get; } =
        GD.Load<PackedScene>("res://Path/To/MyScene.tscn");
    private Node _node;
    private Node _myNode;
    private Node _myScene;
    private Node _myInheritedScene;

    public Game()
    {
        _node = new Node();
        _myNode = MyNode.New().As<Node>();
        // Different than calling new() or MyNode.New().
Instantiated from a PackedScene.
        _myScene = MyScene.Instantiate();
        // Create scene inheriting from MyScene.
        _myInheritedScene =
MyScene.Instantiate(PackedScene.GenEditState.Main);
    }
}
```

Also, scripts will operate a little slower than scenes due to the speed differences between engine and script code. The larger and more complex the node, the more reason there is to build it as a scene.

Named types

Scripts can be registered as a new type within the editor itself. This displays it as a new type in the node or resource creation dialog with an optional icon. This way, the user's ability to use the script is much more streamlined. Rather than having to...

1. Know the base type of the script they would like to use.
2. Create an instance of that base type.
3. Add the script to the node.

With a registered script, the scripted type instead becomes a creation option like the other nodes and resources in the system. The creation dialog even has a search bar to look up the type by name.

There are two systems for registering types:

- Custom Types
    - Editor-only. Typenames are not accessible at runtime.
    - Does not support inherited custom types.
    - An initializer tool. Creates the node with the script. Nothing more.
    - Editor has no type-awareness of the script or its relationship to other engine types or scripts.
    - Allows users to define an icon.
    - Works for all scripting languages because it deals with Script resources in abstract.
    - Set up using EditorPlugin.add_custom_type.
- Script Classes
    - Editor and runtime accessible.
    - Displays inheritance relationships in full.
    - Creates the node with the script, but can also change types or extend the type from the editor.
    - Editor is aware of inheritance relationships between scripts, script classes, and engine C++ classes.
    - Allows users to define an icon.
    - Engine developers must add support for languages manually (both name exposure and runtime accessibility).
    - Godot 3.1+ only.

    - The Editor scans project folders and registers any exposed names for all scripting languages. Each scripting language must implement its own support for exposing this information.

Both methodologies add names to the creation dialog, but script classes, in particular, also allow for users to access the typename without loading the script resource. Creating instances and accessing constants or static methods is viable from anywhere.

With features like these, one may wish their type to be a script without a scene due to the ease of use it grants users. Those developing plugins or creating in-house tools for designers to use will find an easier time of things this way.

On the downside, it also means having to use largely imperative programming.

Performance of Script vs PackedScene

One last aspect to consider when choosing scenes and scripts is execution speed.

As the size of objects increases, the scripts' necessary size to create and initialize them grows much larger. Creating node hierarchies demonstrates this. Each Node's logic could be several hundred lines of code in length.

The code example below creates a new `Node`, changes its name, assigns a script to it, sets its future parent as its owner so it gets saved to disk along with it, and finally adds it as a child of the `Main` node:

GDScript

```
# main.gd
extends Node

func _init():
    var child = Node.new()
    child.name = "Child"
    child.script = preload("child.gd")
    add_child(child)
    child.owner = self
```

C#

```
using Godot;

public partial class Main : Node
{
    public Node Child { get; set; }

    public Main()
    {
        Child = new Node();
        Child.Name = "Child";
        var childID = Child.GetInstanceId();
        Child.SetScript(GD.Load<Script>
("res://Path/To/Child.cs"));
        // SetScript() causes the C# wrapper object to be
disposed, so obtain a new
        // wrapper for the Child node using its instance ID
before proceeding.
        Child = (Node)GodotObject.InstanceFromId(childID);
        AddChild(Child);
        Child.Owner = this;
    }
}
```

Script code like this is much slower than engine-side C++ code. Each instruction makes a call to the scripting API which leads to many "lookups" on the back-end to find the logic to execute.

Scenes help to avoid this performance issue. PackedScene, the base type that scenes inherit from, defines resources that use serialized data to create objects. The engine can process scenes in batches on the back-end and provide much better performance than scripts.

Conclusion

In the end, the best approach is to consider the following:

- If one wishes to create a basic tool that is going to be re-used in several different projects and which people of all skill levels will likely use (including those who don't label themselves as "programmers"), then chances are that it should probably be a script, likely one with a custom name/icon.

- If one wishes to create a concept that is particular to their game, then it should always be a scene. Scenes are easier to track/edit and provide more security than scripts.

- If one would like to give a name to a scene, then they can still sort of do this by declaring a script class and giving it a scene as a constant. The script becomes, in effect, a namespace:

  GDScript

  ```
  # game.gd
  class_name Game # extends RefCounted, so it won't show up in the node creation dialog.
  extends RefCounted

  const MyScene = preload("my_scene.tscn")

  # main.gd
  extends Node
  func _ready():
      add_child(Game.MyScene.instantiate())
  ```

  C#

```
// Game.cs
public partial class Game : RefCounted
{
    public static PackedScene MyScene { get; } =
        GD.Load<PackedScene>("res://Path/To/MyScene.tscn");
}

// Main.cs
public partial class Main : Node
{
    public override void _Ready()
    {
        AddChild(Game.MyScene.Instantiate());
    }
}
```

Autoloads versus regular nodes

Godot offers a feature to automatically load nodes at the root of your project, allowing you to access them globally, that can fulfill the role of a Singleton: Singletons (Autoload). These autoloaded nodes are not freed when you change the scene from code with SceneTree.change_scene_to_file.

In this guide, you will learn when to use the Autoload feature, and techniques you can use to avoid it.

The cutting audio issue

Other engines can encourage the use of creating manager classes, singletons that organize a lot of functionality into a globally accessible object. Godot offers many ways to avoid global state thanks to the node tree and signals.

For example, let's say we are building a platformer and want to collect coins that play a sound effect. There's a node for that: the AudioStreamPlayer. But if we call the `AudioStreamPlayer` while it is already playing a sound, the new sound interrupts the first.

A solution is to code a global, autoloaded sound manager class. It generates a pool of `AudioStreamPlayer` nodes that cycle through as each new request for sound effects comes in. Say we call that class `Sound`, you can use it from anywhere in your project by calling `Sound.play("coin_pickup.ogg")`. This solves the problem in the short term but causes more problems:

1. **Global state**: one object is now responsible for all objects' data. If the `Sound` class has errors or doesn't have an AudioStreamPlayer available, all the nodes calling it can break.
2. **Global access**: now that any object can call `Sound.play(sound_path)` from anywhere, there's no longer an easy way to find the source of a bug.
3. **Global resource allocation**: with a pool of `AudioStreamPlayer` nodes stored from the start, you can either have too few and face bugs, or too many and use more memory than you need.

Note

About global access, the problem is that any code anywhere could pass wrong data to the `Sound` autoload in our example. As a result, the domain to explore to fix the bug spans the entire project.

When you keep code inside a scene, only one or two scripts may be involved in audio.

Contrast this with each scene keeping as many `AudioStreamPlayer` nodes as it needs within itself and all these problems go away:

1. Each scene manages its own state information. If there is a problem with the data, it will only cause issues in that one scene.
2. Each scene accesses only its own nodes. Now, if there is a bug, it's easy to find which node is at fault.
3. Each scene allocates exactly the amount of resources it needs.

Managing shared functionality or data

Another reason to use an Autoload can be that you want to reuse the same method or data across many scenes.

In the case of functions, you can create a new type of `Node` that provides that feature for an individual scene using the class_name keyword in GDScript.

When it comes to data, you can either:

1. Create a new type of Resource to share the data.
2. Store the data in an object to which each node has access, for example using the `owner` property to access the scene's root node.

When you should use an Autoload

GDScript supports the creation of `static` functions using `static func`. When combined with `class_name`, this makes it possible to create libraries of helper functions without having to create an instance to call them. The limitation of static functions is that they can't reference member variables, non-static functions or `self`.

Since Godot 4.1, GDScript also supports `static` variables using `static var`. This means you can now share variables across instances of a class without having to create a separate autoload.

Still, autoloaded nodes can simplify your code for systems with a wide scope. If the autoload is managing its own information and not invading the data of other objects, then it's a great way to create systems that handle broad-scoped tasks. For example, a quest or a dialogue system.

Note

An autoload is *not* necessarily a singleton. Nothing prevents you from instantiating copies of an autoloaded node. An autoload is only a tool that makes a node load automatically as a child of the root of your scene tree, regardless of your game's node structure or which scene you run, e.g. by pressing the `F6` key.

As a result, you can get the autoloaded node, for example an autoload called `Sound`, by calling `get_node("/root/Sound")`.

When and how to avoid using nodes for everything

Nodes are cheap to produce, but even they have their limits. A project may have tens of thousands of nodes all doing things. The more complex their behavior though, the larger the strain each one adds to a project's performance.

Godot provides more lightweight objects for creating APIs which nodes use. Be sure to keep these in mind as options when designing how you wish to build your project's features.

1. Object: The ultimate lightweight object, the original Object must use manual memory management. With that said, it isn't too difficult to create one's own custom data structures, even node structures, that are also lighter than the Node class.

    - **Example:** See the Tree node. It supports a high level of customization for a table of content with an arbitrary number of rows and columns. The data that it uses to generate its visualization though is actually a tree of TreeItem Objects.
    - **Advantages:** Simplifying one's API to smaller scoped objects helps improve its accessibility and improve iteration time. Rather than working with the entire Node library, one creates an abbreviated set of Objects from which a node can generate and manage the appropriate sub-nodes.

    Note

    One should be careful when handling them. One can store an Object into a variable, but these references can become invalid without warning. For example, if the object's creator decides to delete it out of nowhere, this would trigger an error state when one next accesses it.

2. RefCounted: Only a little more complex than Object. They track references to themselves, only deleting loaded memory when no further references to themselves exist. These are useful in the majority of cases where one needs data in a custom class.
    - **Example:** See the FileAccess object. It functions just like a regular Object except that one need not delete it themselves.
    - **Advantages:** same as the Object.
3. Resource: Only slightly more complex than RefCounted. They have the innate ability to serialize/deserialize (i.e. save and load) their object properties to/from Godot resource files.
    - **Example:** Scripts, PackedScene (for scene files), and other types like each of the AudioEffect classes. Each of these can be saved and loaded, therefore they extend from Resource.
    - **Advantages:** Much has already been said on Resource's advantages over traditional data storage methods. In the context of using Resources over Nodes though, their main advantage is in Inspector-compatibility. While nearly as lightweight as Object/RefCounted, they can still display and export properties in the Inspector. This allows them to fulfill a purpose much like sub-Nodes on the usability front, but also improve performance if one plans to have many such Resources/Nodes in their scenes.

Godot interfaces

Often one needs scripts that rely on other objects for features. There are 2 parts to this process:

1. Acquiring a reference to the object that presumably has the features.
2. Accessing the data or logic from the object.

The rest of this tutorial outlines the various ways of doing all this.

Acquiring object references

For all Objects, the most basic way of referencing them is to get a reference to an existing object from another acquired instance.

GDScript

```
var obj = node.object # Property access.
var obj = node.get_object() # Method access.
```

C#

```
GodotObject obj = node.Object; // Property access.
GodotObject obj = node.GetObject(); // Method access.
```

The same principle applies for RefCounted objects. While users often access Node and Resource this way, alternative measures are available.

Instead of property or method access, one can get Resources by load access.

GDScript

```
# If you need an "export const var" (which doesn't exist), use a conditional
# setter for a tool script that checks if it's executing in the editor.
# The `@tool` annotation must be placed at the top of the script.
@tool

# Load resource during scene load.
var preres = preload(path)
# Load resource when program reaches statement.
var res = load(path)

# Note that users load scenes and scripts, by convention, with PascalCase
# names (like typenames), often into constants.
const MyScene = preload("my_scene.tscn") # Static load
const MyScript = preload("my_script.gd")

# This type's value varies, i.e. it is a variable, so it uses snake_case.
@export var script_type: Script

# Must configure from the editor, defaults to null.
@export var const_script: Script:
    set(value):
        if Engine.is_editor_hint():
            const_script = value

# Warn users if the value hasn't been set.
func _get_configuration_warnings():
    if not const_script:
        return ["Must initialize property 'const_script'."]

    return []
```

C#

```
// Tool script added for the sake of the "const [Export]" example.
[Tool]
public MyType
{
    // Property initializations load during Script instancing,
i.e. .new().
    // No "preload" loads during scene load exists in C#.

    // Initialize with a value. Editable at runtime.
    public Script MyScript = GD.Load<Script>
("res://Path/To/MyScript.cs");

    // Initialize with same value. Value cannot be changed.
    public readonly Script MyConstScript = GD.Load<Script>
("res://Path/To/MyScript.cs");

    // Like 'readonly' due to inaccessible setter.
    // But, value can be set during constructor, i.e. MyType().
    public Script MyNoSetScript { get; } = GD.Load<Script>
("res://Path/To/MyScript.cs");

    // If need a "const [Export]" (which doesn't exist), use a
    // conditional setter for a tool script that checks if it's
executing
    // in the editor.
    private PackedScene _enemyScn;

    [Export]
    public PackedScene EnemyScn
    {
        get { return _enemyScn; }
        set
        {
            if (Engine.IsEditorHint())
            {
                _enemyScn = value;
            }
        }
    };

    // Warn users if the value hasn't been set.
    public string[] _GetConfigurationWarnings()
    {
        if (EnemyScn == null)
        {
            return new string[] { "Must initialize property
'EnemyScn'." };
```

```
        }
        return Array.Empty<string>();
    }
}
```

Note the following:

1. There are many ways in which a language can load such resources.
2. When designing how objects will access data, don't forget that one can pass resources around as references as well.
3. Keep in mind that loading a resource fetches the cached resource instance maintained by the engine. To get a new object, one must duplicate an existing reference or instantiate one from scratch with `new()`.

Nodes likewise have an alternative access point: the SceneTree.

GDScript

```
extends Node

# Slow.
func dynamic_lookup_with_dynamic_nodepath():
    print(get_node("Child"))

# Faster. GDScript only.
func dynamic_lookup_with_cached_nodepath():
    print($Child)

# Fastest. Doesn't break if node moves later.
# Note that `@onready` annotation is GDScript-only.
# Other languages must do...
#     var child
#     func _ready():
#         child = get_node("Child")
@onready var child = $Child
func lookup_and_cache_for_future_access():
    print(child)
```

```
# Fastest. Doesn't break if node is moved in the Scene tree dock.
# Node must be selected in the inspector as it's an exported property.
@export var child: Node
func lookup_and_cache_for_future_access():
    print(child)

# Delegate reference assignment to an external source.
# Con: need to perform a validation check.
# Pro: node makes no requirements of its external structure.
#      'prop' can come from anywhere.
var prop
func call_me_after_prop_is_initialized_by_parent():
    # Validate prop in one of three ways.

    # Fail with no notification.
    if not prop:
        return

    # Fail with an error message.
    if not prop:
        printerr("'prop' wasn't initialized")
        return

    # Fail and terminate.
    # NOTE: Scripts run from a release export template don't run `assert`s.
    assert(prop, "'prop' wasn't initialized")

# Use an autoload.
# Dangerous for typical nodes, but useful for true singleton nodes
# that manage their own data and don't interfere with other objects.
func reference_a_global_autoloaded_variable():
    print(globals)
    print(globals.prop)
    print(globals.my_getter())
```

C#

```
using Godot;
using System;
using System.Diagnostics;

public class MyNode : Node
{
    // Slow
    public void DynamicLookupWithDynamicNodePath()
    {
        GD.Print(GetNode("Child"));
    }

    // Fastest. Lookup node and cache for future access.
    // Doesn't break if node moves later.
    private Node _child;
    public void _Ready()
    {
        _child = GetNode("Child");
    }
    public void LookupAndCacheForFutureAccess()
    {
        GD.Print(_child);
    }

    // Delegate reference assignment to an external source.
    // Con: need to perform a validation check.
    // Pro: node makes no requirements of its external
structure.
    //      'prop' can come from anywhere.
    public object Prop { get; set; }
    public void CallMeAfterPropIsInitializedByParent()
    {
        // Validate prop in one of three ways.

        // Fail with no notification.
        if (prop == null)
        {
            return;
        }

        // Fail with an error message.
        if (prop == null)
        {
            GD.PrintErr("'Prop' wasn't initialized");
            return;
        }

        // Fail with an exception.
        if (prop == null)
        {
            throw new InvalidOperationException("'Prop' wasn't
initialized.");
        }

        // Fail and terminate.
        // Note: Scripts run from a release export template
don't run `Debug.Assert`s.
        Debug.Assert(Prop, "'Prop' wasn't initialized");
    }

    // Use an autoload.
    // Dangerous for typical nodes, but useful for true
singleton nodes
    // that manage their own data and don't interfere with
other objects.
    public void ReferenceAGlobalAutoloadedVariable()
    {
        MyNode globals = GetNode<MyNode>("/root/Globals");
        GD.Print(globals);
        GD.Print(globals.Prop);
        GD.Print(globals.MyGetter());
    }
};
```

Accessing data or logic from an object

Godot's scripting API is duck-typed. This means that if a script executes an operation, Godot doesn't validate that it supports the operation by **type**. It instead checks that the object **implements** the individual method.

For example, the CanvasItem class has a `visible` property. All properties exposed to the scripting API are in fact a setter and getter pair bound to a name. If one tried to access CanvasItem.visible, then Godot would do the following checks, in order:

- If the object has a script attached, it will attempt to set the property through the script. This leaves open the opportunity for scripts to override a property defined on a base object by overriding the setter method for the property.
- If the script does not have the property, it performs a HashMap lookup in the ClassDB for the "visible" property against the CanvasItem class and all of its inherited types. If found, it will call the bound setter or getter. For more information about HashMaps, see the data preferences docs.
- If not found, it does an explicit check to see if the user wants to access the "script" or "meta" properties.
- If not, it checks for a `_set`/`_get` implementation (depending on type of access) in the CanvasItem and its inherited types. These methods can execute logic that gives the impression that the Object has a property. This is also the case with the `_get_property_list` method.
  - Note that this happens even for non-legal symbol names, such as names starting with a digit or containing a slash.

As a result, this duck-typed system can locate a property either in the script, the object's class, or any class that object inherits, but only for things which extend Object.

Godot provides a variety of options for performing runtime checks on these accesses:

- A duck-typed property access. These will be property checks (as described above). If the operation isn't supported by the object, execution will halt.

  GDScript

```
# All Objects have duck-typed get, set, and call wrapper
methods.
get_parent().set("visible", false)

# Using a symbol accessor, rather than a string in the
method call,
# will implicitly call the `set` method which, in turn,
calls the
# setter method bound to the property through the property
lookup
# sequence.
get_parent().visible = false

# Note that if one defines a _set and _get that describe a
property's
# existence, but the property isn't recognized in any
_get_property_list
# method, then the set() and get() methods will work, but
the symbol
# access will claim it can't find the property.
```

C#

```
// All Objects have duck-typed Get, Set, and Call wrapper
methods.
GetParent().Set("visible", false);

// C# is a static language, so it has no dynamic symbol
access, e.g.
// `GetParent().Visible = false` won't work.
```

- A method check. In the case of CanvasItem.visible, one can access the methods, `set_visible` and `is_visible` like any other method.

  GDScript

```
var child = get_child(0)

# Dynamic lookup.
child.call("set_visible", false)
```

```
# Symbol-based dynamic lookup.
# GDScript aliases this into a 'call' method behind the
scenes.
child.set_visible(false)

# Dynamic lookup, checks for method existence first.
if child.has_method("set_visible"):
    child.set_visible(false)

# Cast check, followed by dynamic lookup.
# Useful when you make multiple "safe" calls knowing that
the class
# implements them all. No need for repeated checks.
# Tricky if one executes a cast check for a user-defined
type as it
# forces more dependencies.
if child is CanvasItem:
    child.set_visible(false)
    child.show_on_top = true

# If one does not wish to fail these checks without
notifying users,
# one can use an assert instead. These will trigger runtime
errors
# immediately if not true.
assert(child.has_method("set_visible"))
assert(child.is_in_group("offer"))
assert(child is CanvasItem)

# Can also use object labels to imply an interface, i.e.
assume it
# implements certain methods.
# There are two types, both of which only exist for Nodes:
Names and
# Groups.

# Assuming...
# A "Quest" object exists and 1) that it can "complete" or
"fail" and
# that it will have text available before and after each
state...

# 1. Use a name.
```

```
var quest = $Quest
print(quest.text)
quest.complete() # or quest.fail()
print(quest.text) # implied new text content

# 2. Use a group.
for a_child in get_children():
    if a_child.is_in_group("quest"):
        print(quest.text)
        quest.complete() # or quest.fail()
        print(quest.text) # implied new text content

# Note that these interfaces are project-specific
conventions the team
# defines (which means documentation! But maybe worth it?).
# Any script that conforms to the documented "interface" of
the name or
# group can fill in for it.
```

C#

```
Node child = GetChild(0);

// Dynamic lookup.
child.Call("SetVisible", false);

// Dynamic lookup, checks for method existence first.
if (child.HasMethod("SetVisible"))
{
    child.Call("SetVisible", false);
}

// Use a group as if it were an "interface", i.e. assume it
implements
// certain methods.
// Requires good documentation for the project to keep it
reliable
// (unless you make editor tools to enforce it at editor
time).
// Note, this is generally not as good as using an actual
interface in
// C#, but you can't set C# interfaces from the editor
```

```
since they are
// language-level features.
if (child.IsInGroup("Offer"))
{
    child.Call("Accept");
    child.Call("Reject");
}

// Cast check, followed by static lookup.
CanvasItem ci = GetParent() as CanvasItem;
if (ci != null)
{
    ci.SetVisible(false);

    // useful when you need to make multiple safe calls to
the class
    ci.ShowOnTop = true;
}

// If one does not wish to fail these checks without
notifying users,
// one can use an assert instead. These will trigger
runtime errors
// immediately if not true.
Debug.Assert(child.HasMethod("set_visible"));
Debug.Assert(child.IsInGroup("offer"));
Debug.Assert(CanvasItem.InstanceHas(child));

// Can also use object labels to imply an interface, i.e.
assume it
// implements certain methods.
// There are two types, both of which only exist for Nodes:
Names and
// Groups.

// Assuming...
// A "Quest" object exists and 1) that it can "Complete" or
"Fail" and
// that it will have Text available before and after each
state...

// 1. Use a name.
Node quest = GetNode("Quest");
```

```
GD.Print(quest.Get("Text"));
quest.Call("Complete"); // or "Fail".
GD.Print(quest.Get("Text")); // Implied new text content.

// 2. Use a group.
foreach (Node AChild in GetChildren())
{
    if (AChild.IsInGroup("quest"))
    {
      GD.Print(quest.Get("Text"));
      quest.Call("Complete"); // or "Fail".
      GD.Print(quest.Get("Text")); // Implied new text
content.
    }
}

// Note that these interfaces are project-specific
conventions the team
// defines (which means documentation! But maybe worth
it?).
// Any script that conforms to the documented "interface"
of the
// name or group can fill in for it. Also note that in C#,
these methods
// will be slower than static accesses with traditional
interfaces.
```

- Outsource the access to a Callable. These may be useful in cases where one needs the max level of freedom from dependencies. In this case, one relies on an external context to setup the method.

GDScript

```
# child.gd
extends Node
var fn = null

func my_method():
    if fn:
        fn.call()
```

```
# parent.gd
extends Node

@onready var child = $Child

func _ready():
    child.fn = print_me
    child.my_method()

func print_me():
    print(name)
```

C#

```
// Child.cs
using Godot;

public partial class Child : Node
{
    public Callable? Callable { get; set; }

    public void MyMethod()
    {
        Callable?.Call();
    }
}

// Parent.cs
using Godot;

public partial class Parent : Node
{
    private Child _child;

    public void _Ready()
    {
        _child = GetNode<Child>("Child");
        _child.Callable = Callable.From(PrintMe);
        _child.MyMethod();
    }
    public void PrintMe()
    {
        GD.Print(Name);
    }
}
```

These strategies contribute to Godot's flexible design. Between them, users have a breadth of tools to meet their specific needs.

Godot notifications

Every Object in Godot implements a _notification method. Its purpose is to allow the Object to respond to a variety of engine-level callbacks that may relate to it. For example, if the engine tells a CanvasItem to "draw", it will call `_notification(NOTIFICATION_DRAW)`.

Some of these notifications, like draw, are useful to override in scripts. So much so that Godot exposes many of them with dedicated functions:

- `_ready()`: `NOTIFICATION_READY`
- `_enter_tree()`: `NOTIFICATION_ENTER_TREE`
- `_exit_tree()`: `NOTIFICATION_EXIT_TREE`
- `_process(delta)`: `NOTIFICATION_PROCESS`
- `_physics_process(delta)`: `NOTIFICATION_PHYSICS_PROCESS`
- `_draw()`: `NOTIFICATION_DRAW`

What users might *not* realize is that notifications exist for types other than Node alone, for example:

- Object::NOTIFICATION_POSTINITIALIZE: a callback that triggers during object initialization. Not accessible to scripts.
- Object::NOTIFICATION_PREDELETE: a callback that triggers before the engine deletes an Object, i.e. a "destructor".

And many of the callbacks that *do* exist in Nodes don't have any dedicated methods, but are still quite useful.

- Node::NOTIFICATION_PARENTED: a callback that triggers anytime one adds a child node to another node.
- Node::NOTIFICATION_UNPARENTED: a callback that triggers anytime one removes a child node from another node.

One can access all these custom notifications from the universal `_notification()` method.

Note

Methods in the documentation labeled as "virtual" are also intended to be overridden by scripts.

A classic example is the _init method in Object. While it has no `NOTIFICATION_*` equivalent, the engine still calls the method. Most languages (except C#) rely on it as a constructor.

So, in which situation should one use each of these notifications or virtual functions?

_process vs. _physics_process vs. *_input

Use `_process()` when one needs a framerate-dependent delta time between frames. If code that updates object data needs to update as often as possible, this is the right place. Recurring logic checks and data caching often execute here, but it comes down to the frequency at which one needs the evaluations to update. If they don't need to execute every frame, then implementing a Timer-timeout loop is another option.

GDScript

```
# Allows for recurring operations that don't trigger script logic
# every frame (or even every fixed frame).
func _ready():
    var timer = Timer.new()
    timer.autostart = true
    timer.wait_time = 0.5
    add_child(timer)
    timer.timeout.connect(func():
        print("This block runs every 0.5 seconds")
    )
```

C#

```
using Godot;

public partial class MyNode : Node
{
    // Allows for recurring operations that don't trigger script logic
    // every frame (or even every fixed frame).
    public override void _Ready()
    {
        var timer = new Timer();
        timer.Autostart = true;
        timer.WaitTime = 0.5;
        AddChild(timer);
        timer.Timeout += () => GD.Print("This block runs every 0.5 seconds");
    }
}
```

Use `_physics_process()` when one needs a framerate-independent delta time between frames. If code needs consistent updates over time, regardless of how fast or slow time advances, this is the right place. Recurring kinematic and object transform operations should execute here.

While it is possible, to achieve the best performance, one should avoid making input checks during these callbacks. `_process()` and `_physics_process()` will trigger at every opportunity (they do not "rest" by default). In contrast, `*_input()` callbacks will trigger only on frames in which the engine has actually detected the input.

One can check for input actions within the input callbacks just the same. If one wants to use delta time, one can fetch it from the related delta time methods as needed.

GDScript

```gdscript
# Called every frame, even when the engine detects no input.
func _process(delta):
    if Input.is_action_just_pressed("ui_select"):
        print(delta)

# Called during every input event.
func _unhandled_input(event):
    match event.get_class():
        "InputEventKey":
            if Input.is_action_just_pressed("ui_accept"):
                print(get_process_delta_time())
```

C#

```csharp
using Godot;

public partial class MyNode : Node
{

    // Called every frame, even when the engine detects no
input.
    public void _Process(double delta)
    {
        if (Input.IsActionJustPressed("ui_select"))
            GD.Print(delta);
    }

    // Called during every input event. Equally true for
_input().
    public void _UnhandledInput(InputEvent @event)
    {
        switch (@event)
        {
            case InputEventKey:
                if (Input.IsActionJustPressed("ui_accept"))
                    GD.Print(GetProcessDeltaTime());
                    break;
            }
        }

}
```

_init vs. initialization vs. export

If the script initializes its own node subtree, without a scene, that code should execute in `_init()`. Other property or SceneTree-independent initializations should also run here.

Note

The C# equivalent to GDScript's `_init()` method is the constructor.

`_init()` triggers before `_enter_tree()` or `_ready()`, but after a script creates and initializes its properties. When instantiating a scene, property values will set up according to the following sequence:

1. **Initial value assignment:** the property is assigned its initialization value, or its default value if one is not specified. If a setter exists, it is not used.
2. **``_init()`` assignment:** the property's value is replaced by any assignments made in `_init()`, triggering the setter.
3. **Exported value assignment:** an exported property's value is again replaced by any value set in the Inspector, triggering the setter.

GDScript

```
# test is initialized to "one", without triggering the setter.
@export var test: String = "one":
    set(value):
        test = value + "!"
```

```
func _init():
    # Triggers the setter, changing test's value from "one" to
"two!".
    test = "two"

# If someone sets test to "three" from the Inspector, it would
trigger
# the setter, changing test's value from "two!" to "three!".
```

C#

```
using Godot;

public partial class MyNode : Node
{
    private string _test = "one";

    [Export]
    public string Test
    {
        get { return _test; }
        set { _test = $"{value}!"; }
    }

    public MyNode()
    {
        // Triggers the setter, changing _test's value from
"one" to "two!".
        Test = "two";
    }

    // If someone sets Test to "three" in the Inspector, it
would trigger
    // the setter, changing _test's value from "two!" to
"three!".
}
```

As a result, instantiating a script versus a scene may affect both the initialization *and* the number of times the engine calls the setter.

_ready vs. _enter_tree vs. NOTIFICATION_PARENTED

When instantiating a scene connected to the first executed scene, Godot will instantiate nodes down the tree (making `_init()` calls) and build the tree going downwards from the root. This causes `_enter_tree()` calls to cascade down the tree. Once the tree is complete, leaf nodes call `_ready`. A node will call this method once all child nodes have finished calling theirs. This then causes a reverse cascade going up back to the tree's root.

When instantiating a script or a standalone scene, nodes are not added to the SceneTree upon creation, so no `_enter_tree()` callbacks trigger. Instead, only the `_init()` call occurs. When the scene is added to the SceneTree, the `_enter_tree()` and `_ready()` calls occur.

If one needs to trigger behavior that occurs as nodes parent to another, regardless of whether it occurs as part of the main/active scene or not, one can use the PARENTED notification. For example, here is a snippet that connects a node's method to a custom signal on the parent node without failing. Useful on data-centric nodes that one might create at runtime.

GDScript

```
extends Node

var parent_cache

func connection_check():
    return parent_cache.has_user_signal("interacted_with")

func _notification(what):
    match what:
        NOTIFICATION_PARENTED:
            parent_cache = get_parent()
            if connection_check():

parent_cache.interacted_with.connect(_on_parent_interacted_with
)
        NOTIFICATION_UNPARENTED:
            if connection_check():

parent_cache.interacted_with.disconnect(_on_parent_interacted_w
ith)

func _on_parent_interacted_with():
    print("I'm reacting to my parent's interaction!")
```

C#

```
using Godot;

public partial class MyNode : Node
{
    private Node _parentCache;

    public bool ConnectionCheck()
    {
        return _parentCache.HasUserSignal("InteractedWith");
    }

    public void _Notification(int what)
    {
        switch (what)
        {
            case NotificationParented:
                _parentCache = GetParent();
                if (ConnectionCheck())
                {
                    _parentCache.Connect("InteractedWith",
Callable.From(OnParentInteractedWith));
                }
                break;
            case NotificationUnparented:
                if (ConnectionCheck())
                {
                    _parentCache.Disconnect("InteractedWith",
Callable.From(OnParentInteractedWith));
                }
                break;
        }
    }

    private void OnParentInteractedWith()
    {
        GD.Print("I'm reacting to my parent's interaction!");
    }
}
```

Data preferences

Ever wondered whether one should approach problem X with data structure Y or Z? This article covers a variety of topics related to these dilemmas.

Note

This article makes references to "[something]-time" operations. This terminology comes from algorithm analysis' Big O Notation [https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/].

Long-story short, it describes the worst-case scenario of runtime length. In laymen's terms:

"As the size of a problem domain increases, the runtime length of the algorithm..."

- Constant-time, `O(1)`: "...does not increase."
- Logarithmic-time, `O(log n)`: "...increases at a slow rate."
- Linear-time, `O(n)`: "...increases at the same rate."
- Etc.

Imagine if one had to process 3 million data points within a single frame. It would be impossible to craft the feature with a linear-time algorithm since the sheer size of the data would increase the runtime far beyond the time allotted. In comparison, using a constant-time algorithm could handle the operation without issue.

By and large, developers want to avoid engaging in linear-time operations as much as possible. But, if one keeps the scale of a linear-time operation small, and if one does not need to perform the operation often, then it may be acceptable. Balancing these requirements and choosing the right algorithm / data structure for the job is part of what makes programmers' skills valuable.

Array vs. Dictionary vs. Object

Godot stores all variables in the scripting API in the Variant class. Variants can store Variant-compatible data structures such as Array and Dictionary as well as Objects.

Godot implements Array as a `Vector<Variant>`. The engine stores the Array contents in a contiguous section of memory, i.e. they are in a row adjacent to each other.

Note

For those unfamiliar with C++, a Vector is the name of the array object in traditional C++ libraries. It is a "templated" type, meaning that its records can only contain a particular type (denoted by angled brackets). So, for example, a PackedStringArray would be something like a `Vector<String>`.

Contiguous memory stores imply the following operation performance:

- **Iterate:** Fastest. Great for loops.
    - Op: All it does is increment a counter to get to the next record.
- **Insert, Erase, Move:** Position-dependent. Generally slow.
    - Op: Adding/removing/moving content involves moving the adjacent records over (to make room / fill space).

    - Fast add/remove *from the end*.
    - Slow add/remove *from an arbitrary position*.
    - Slowest add/remove *from the front*.
    - If doing many inserts/removals *from the front*, then...

        1. invert the array.
        2. do a loop which executes the Array changes *at the end*.
        3. re-invert the array.

      This makes only 2 copies of the array (still constant time, but slow) versus copying roughly 1/2 of the array, on average, N times (linear time).

- **Get, Set:** Fastest *by position*. E.g. can request 0th, 2nd, 10th record, etc. but cannot specify which record you want.
    - Op: 1 addition operation from array start position up to desired index.
- **Find:** Slowest. Identifies the index/position of a value.
    - Op: Must iterate through array and compare values until one finds a match.
        - Performance is also dependent on whether one needs an exhaustive search.
    - If kept ordered, custom search operations can bring it to logarithmic time (relatively fast). Laymen users won't be comfortable with this though. Done by re-sorting the Array after every edit and writing an ordered-aware search algorithm.

Godot implements Dictionary as an `OrderedHashMap<Variant, Variant>`. The engine stores a small array (initialized to 2^3 or 8 records) of key-value pairs. When one attempts to access a value, they provide it a key. It then *hashes* the key, i.e. converts it into a number. The "hash" is used to calculate the index into the array. As an array, the OHM then has a quick lookup within the "table" of keys mapped to values. When the HashMap becomes too full, it increases to the next power of 2 (so, 16 records, then 32, etc.) and rebuilds the structure.

Hashes are to reduce the chance of a key collision. If one occurs, the table must recalculate another index for the value that takes the previous position into account. In all, this results in constant-time access to all records at the expense of memory and some minor operational efficiency.

1. Hashing every key an arbitrary number of times.
    - Hash operations are constant-time, so even if an algorithm must do more than one, as long as the number of hash calculations doesn't become too dependent on the density of the table, things will stay fast. Which leads to...
2. Maintaining an ever-growing size for the table.
    - HashMaps maintain gaps of unused memory interspersed in the table on purpose to reduce hash collisions and maintain the speed of accesses. This is why it constantly increases in size exponentially by powers of 2.

As one might be able to tell, Dictionaries specialize in tasks that Arrays do not. An overview of their operational details is as follows:

- **Iterate:** Fast.

    - Op: Iterate over the map's internal vector of hashes. Return each key. Afterwards, users then use the key to jump to and return the desired value.
- **Insert, Erase, Move:** Fastest.
    - Op: Hash the given key. Do 1 addition operation to look up the appropriate value (array start + offset). Move is two of these (one insert, one erase). The map must do some maintenance to preserve its capabilities:
        - update ordered List of records.
        - determine if table density mandates a need to expand table capacity.
    - The Dictionary remembers in what order users inserted its keys. This enables it to execute reliable iterations.
- **Get, Set:** Fastest. Same as a lookup *by key*.
    - Op: Same as insert/erase/move.
- **Find:** Slowest. Identifies the key of a value.
    - Op: Must iterate through records and compare the value until a match is found.
    - Note that Godot does not provide this feature out-of-the-box (because they aren't meant for this task).

Godot implements Objects as stupid, but dynamic containers of data content. Objects query data sources when posed questions. For example, to answer the question, "do you have a property called, 'position'?", it might ask its script or the ClassDB. One can find more information about what objects are and how they work in the Applying object-oriented principles in Godot article.

The important detail here is the complexity of the Object's task. Every time it performs one of these multi-source queries, it runs through *several* iteration loops and HashMap lookups. What's more, the queries are linear-time operations dependent on the Object's inheritance hierarchy size. If the class the Object queries (its current class) doesn't find anything, the request defers to the next base class, all the way up until the original Object class. While these are each fast operations in isolation, the fact that it must make so many checks is what makes them slower than both of the alternatives for looking up data.

Note

When developers mention how slow the scripting API is, it is this chain of queries they refer to. Compared to compiled C++ code where the application knows exactly where to go to find anything, it is inevitable that scripting API operations will take much longer. They must locate the source of any relevant data before they can attempt to access it.

The reason GDScript is slow is because every operation it performs passes through this system.

C# can process some content at higher speeds via more optimized bytecode. But, if the C# script calls into an engine class' content or if the script tries to access something external to it, it will go through this pipeline.

NativeScript C++ goes even further and keeps everything internal by default. Calls into external structures will go through the scripting API. In NativeScript C++, registering methods to expose them to the scripting API is a manual task. It is at this point that external, non-C++ classes will use the API to locate them.

So, assuming one extends from Reference to create a data structure, like an Array or Dictionary, why choose an Object over the other two options?

1. **Control:** With objects comes the ability to create more sophisticated structures. One can layer abstractions over the data to ensure the external API doesn't change in response to internal data structure changes. What's more, Objects can have signals, allowing for reactive behavior.
2. **Clarity:** Objects are a reliable data source when it comes to the data that scripts and engine classes define for them. Properties may not hold the values one expects, but one doesn't need to worry about whether the property exists in the first place.
3. **Convenience:** If one already has a similar data structure in mind, then extending from an existing class makes the task of building the data structure much easier. In comparison, Arrays and Dictionaries don't fulfill all use cases one might have.

Objects also give users the opportunity to create even more specialized data structures. With it, one can design their own List, Binary Search Tree, Heap, Splay Tree, Graph, Disjoint Set, and any host of other options.

"Why not use Node for tree structures?" one might ask. Well, the Node class contains things that won't be relevant to one's custom data structure. As such, it can be helpful to construct one's own node type when building tree structures.

GDScript

```
extends Object
class_name TreeNode

var _parent: TreeNode = null
var _children := []

func _notification(p_what):
    match p_what:
        NOTIFICATION_PREDELETE:
            # Destructor.
            for a_child in _children:
                a_child.free()
```

C#

```
using Godot;
using System.Collections.Generic;

// Can decide whether to expose getters/setters for properties later
public partial class TreeNode : GodotObject
{
    private TreeNode _parent = null;

    private List<TreeNode> _children = new();

    public override void _Notification(int what)
    {
        switch (what)
        {
            case NotificationPredelete:
                foreach (TreeNode child in _children)
                {
                    node.Free();
                }
                break;
        }
    }
}
```

From here, one can then create their own structures with specific features, limited only by their imagination.

Enumerations: int vs. string

Most languages offer an enumeration type option. GDScript is no different, but unlike most other languages, it allows one to use either integers or strings for the enum values (the latter only when using the `@export_enum` annotation in GDScript). The question then arises, "which should one use?"

The short answer is, "whichever you are more comfortable with." This is a feature specific to GDScript and not Godot scripting in general; The languages prioritizes usability over performance.

On a technical level, integer comparisons (constant-time) will happen faster than string comparisons (linear-time). If one wants to keep up other languages' conventions though, then one should use integers.

The primary issue with using integers comes up when one wants to *print* an enum value. As integers, attempting to print `MY_ENUM` will print `5` or what-have-you, rather than something like `"MyEnum"`. To print an integer enum, one would have to write a Dictionary that maps the corresponding string value for each enum.

If the primary purpose of using an enum is for printing values and one wishes to group them together as related concepts, then it makes sense to use them as strings. That way, a separate data structure to execute on the printing is unnecessary.

AnimatedTexture vs. AnimatedSprite2D vs. AnimationPlayer vs. AnimationTree

Under what circumstances should one use each of Godot's animation classes? The answer may not be immediately clear to new Godot users.

AnimatedTexture is a texture that the engine draws as an animated loop rather than a static image. Users can manipulate...

1. the rate at which it moves across each section of the texture (FPS).
2. the number of regions contained within the texture (frames).

Godot's RenderingServer then draws the regions in sequence at the prescribed rate. The good news is that this involves no extra logic on the part of the engine. The bad news is that users have very little control.

Also note that AnimatedTexture is a Resource unlike the other Node objects discussed here. One might create a Sprite2D node that uses AnimatedTexture as its texture. Or (something the others can't do) one could add AnimatedTextures as tiles in a TileSet and integrate it with a TileMapLayer for many auto-animating backgrounds that all render in a single batched draw call.

The AnimatedSprite2D node, in combination with the SpriteFrames resource, allows one to create a variety of animation sequences through spritesheets, flip between animations, and control their speed, regional offset, and orientation. This makes them well-suited to controlling 2D frame-based animations.

If one needs to trigger other effects in relation to animation changes (for example, create particle effects, call functions, or manipulate other peripheral elements besides the frame-based animation), then one will need to use an AnimationPlayer node in conjunction with the AnimatedSprite2D.

AnimationPlayers are also the tool one will need to use if they wish to design more complex 2D animation systems, such as...

1. **Cut-out animations:** editing sprites' transforms at runtime.
2. **2D Mesh animations:** defining a region for the sprite's texture and rigging a skeleton to it. Then one animates the bones which stretch and bend the texture in proportion to the bones' relationships to each other.
3. A mix of the above.

While one needs an AnimationPlayer to design each of the individual animation sequences for a game, it can also be useful to combine animations for blending, i.e. enabling smooth transitions between these animations. There may also be a hierarchical structure between animations that one plans out for their object. These are the cases where the AnimationTree shines. One can find an in-depth guide on using the AnimationTree here.

Logic preferences

Ever wondered whether one should approach problem X with strategy Y or Z? This article covers a variety of topics related to these dilemmas.

Adding nodes and changing properties: which first?

When initializing nodes from a script at runtime, you may need to change properties such as the node's name or position. A common dilemma is, when should you change those values?

It is the best practice to change values on a node before adding it to the scene tree. Some property's setters have code to update other corresponding values, and that code can be slow! For most cases, this code has no impact on your game's performance, but in heavy use cases such as procedural generation, it can bring your game to a crawl.

For these reasons, it is always a best practice to set the initial values of a node before adding it to the scene tree.

Loading vs. preloading

In GDScript, there exists the global preload method. It loads resources as early as possible to front-load the "loading" operations and avoid loading resources while in the middle of performance-sensitive code.

Its counterpart, the load method, loads a resource only when it reaches the load statement. That is, it will load a resource in-place which can cause slowdowns when it occurs in the middle of sensitive processes. The `load()` function is also an alias for ResourceLoader.load(path) which is accessible to *all* scripting languages.

So, when exactly does preloading occur versus loading, and when should one use either? Let's see an example:

GDScript

```
# my_buildings.gd
extends Node

# Note how constant scripts/scenes have a different naming
scheme than
# their property variants.

# This value is a constant, so it spawns when the Script object
loads.
# The script is preloading the value. The advantage here is
that the editor
# can offer autocompletion since it must be a static path.
const BuildingScn = preload("res://building.tscn")

# 1. The script preloads the value, so it will load as a
dependency
#    of the 'my_buildings.gd' script file. But, because this is
a
#    property rather than a constant, the object won't copy the
preloaded
#    PackedScene resource into the property until the script
instantiates
#    with .new().
#
# 2. The preloaded value is inaccessible from the Script object
alone. As
#    such, preloading the value here actually does not benefit
anyone.
#
```

```
# 3. Because the user exports the value, if this script stored
on
#    a node in a scene file, the scene instantiation code will
overwrite the
#    preloaded initial value anyway (wasting it). It's usually
better to
#    provide null, empty, or otherwise invalid default values
for exports.
#
# 4. It is when one instantiates this script on its own with
.new() that
#    one will load "office.tscn" rather than the exported
value.
@export var a_building : PackedScene = preload("office.tscn")

# Uh oh! This results in an error!
# One must assign constant values to constants. Because `load`
performs a
# runtime lookup by its very nature, one cannot use it to
initialize a
# constant.
const OfficeScn = load("res://office.tscn")

# Successfully loads and only when one instantiates the script!
Yay!
var office_scn = load("res://office.tscn")
```

C#

```
using Godot;

// C# and other languages have no concept of "preloading".
public partial class MyBuildings : Node
{
    //This is a read-only field, it can only be assigned when
it's declared or during a constructor.
    public readonly PackedScene Building =
ResourceLoader.Load<PackedScene>("res://building.tscn");

    public PackedScene ABuilding;

    public override void _Ready()
    {
        // Can assign the value during initialization.
        ABuilding = GD.Load<PackedScene>("res://Office.tscn");
    }
}
```

Preloading allows the script to handle all the loading the moment one loads the script. Preloading is useful, but there are also times when one doesn't wish for it. To distinguish these situations, there are a few things one can consider:

1. If one cannot determine when the script might load, then preloading a resource, especially a scene or script, could result in further loads one does not expect. This could lead to unintentional, variable-length load times on top of the original script's load operations.
2. If something else could replace the value (like a scene's exported initialization), then preloading the value has no meaning. This point isn't a significant factor if one intends to always create the script on its own.
3. If one wishes only to 'import' another class resource (script or scene), then using a preloaded constant is often the best course of action. However, in exceptional cases, one may wish not to do this:
    1. If the 'imported' class is liable to change, then it should be a property instead, initialized either using an `@export` or a `load()` (and perhaps not even initialized until later).
    2. If the script requires a great many dependencies, and one does not wish to consume so much memory, then one may wish to, load and unload various dependencies at runtime as circumstances change. If one preloads resources into constants, then the only way to unload these resources would be to unload the entire script. If they are instead loaded properties, then one can set them to `null` and remove all references to the resource entirely (which, as a

RefCounted-extending type, will cause the resources to delete themselves from memory).

Large levels: static vs. dynamic

If one is creating a large level, which circumstances are most appropriate? Should they create the level as one static space? Or should they load the level in pieces and shift the world's content as needed?

Well, the simple answer is, "when the performance requires it." The dilemma associated with the two options is one of the age-old programming choices: does one optimize memory over speed, or vice versa?

The naive answer is to use a static level that loads everything at once. But, depending on the project, this could consume a large amount of memory. Wasting users' RAM leads to programs running slow or outright crashing from everything else the computer tries to do at the same time.

No matter what, one should break larger scenes into smaller ones (to aid in reusability of assets). Developers can then design a node that manages the creation/loading and deletion/unloading of resources and nodes in real-time. Games with large and varied environments or procedurally generated elements often implement these strategies to avoid wasting memory.

On the flip side, coding a dynamic system is more complex, i.e. uses more programmed logic, which results in opportunities for errors and bugs. If one isn't careful, they can develop a system that bloats the technical debt of the application.

As such, the best options would be...

1. To use a static level for smaller games.

2. If one has the time/resources on a medium/large game, create a library or plugin that can code the management of nodes and resources. If refined over time, so as to improve usability and stability, then it could evolve into a reliable tool across projects.
3. Code the dynamic logic for a medium/large game because one has the coding skills, but not the time or resources to refine the code (game's gotta get done). Could potentially refactor later to outsource the code into a plugin.

For an example of the various ways one can swap scenes around at runtime, please see the "Change scenes manually" documentation.

Project organization

Introduction

Since Godot has no restrictions on project structure or filesystem usage, organizing files when learning the engine can seem challenging. This tutorial suggests a workflow which should be a good starting point. We will also cover using version control with Godot.

Organization

Godot is scene-based in nature, and uses the filesystem as-is, without metadata or an asset database.

Unlike other engines, many resources are contained within the scene itself, so the amount of files in the filesystem is considerably lower.

Considering that, the most common approach is to group assets as close to scenes as possible; when a project grows, it makes it more maintainable.

As an example, one can usually place into a single folder their basic assets, such as sprite images, 3D model meshes, materials, and music, etc. They can then use a separate folder to store built levels that use them.

```
/project.godot
/docs/.gdignore  # See "Ignoring specific folders" below
/docs/learning.html
/models/town/house/house.dae
/models/town/house/window.png
/models/town/house/door.png
```

```
/characters/player/cubio.dae
/characters/player/cubio.png
/characters/enemies/goblin/goblin.dae
/characters/enemies/goblin/goblin.png
/characters/npcs/suzanne/suzanne.dae
/characters/npcs/suzanne/suzanne.png
/levels/riverdale/riverdale.scn
```

Style guide

For consistency across projects, we recommend following these guidelines:

- Use **snake_case** for folder and file names (with the exception of C# scripts). This sidesteps case sensitivity issues that can crop up after exporting a project on Windows. C# scripts are an exception to this rule, as the convention is to name them after the class name which should be in PascalCase.
- Use **PascalCase** for node names, as this matches built-in node casing.
- In general, keep third-party resources in a top-level `addons/` folder, even if they aren't editor plugins. This makes it easier to track which files are third-party. There are some exceptions to this rule; for instance, if you use third-party game assets for a character, it makes more sense to include them within the same folder as the character scenes and scripts.

Importing

Godot versions prior to 3.0 did the import process from files outside the project. While this can be useful in large projects, it resulted in an organization hassle for most developers.

Because of this, assets are now transparently imported from within the project folder.

Ignoring specific folders

To prevent Godot from importing files contained in a specific folder, create an empty file called `.gdignore` in the folder (the leading `.` is required). This can be useful to speed up the initial project importing.

Note

To create a file whose name starts with a dot on Windows, place a dot at both the beginning and end of the filename (".gdignore."). Windows will automatically remove the trailing dot when you confirm the name.

Alternatively, you can use a text editor such as Notepad++ or use the following command in a command prompt: `type nul > .gdignore`

Once the folder is ignored, resources in that folder can't be loaded anymore using the `load()` and `preload()` methods. Ignoring a folder will also automatically hide it from the FileSystem dock, which can be useful to reduce clutter.

Note that the `.gdignore` file's contents are ignored, which is why the file should be empty. It does not support patterns like `.gitignore` files do.

Case sensitivity

Windows and recent macOS versions use case-insensitive filesystems by default, whereas Linux distributions use a case-sensitive filesystem by default. This can cause issues after exporting a project, since Godot's PCK virtual filesystem is case-sensitive. To avoid this, it's recommended to stick to `snake_case` naming for all files in the project (and lowercase characters in general).

Note

You can break this rule when style guides say otherwise (such as the C# style guide). Still, be consistent to avoid mistakes.

On Windows 10, to further avoid mistakes related to case sensitivity, you can also make the project folder case-sensitive. After enabling the Windows Subsystem for Linux feature, run the following command in a PowerShell window:

```
# To enable case-sensitivity:
fsutil file setcasesensitiveinfo <path to project folder>
enable

# To disable case-sensitivity:
fsutil file setcasesensitiveinfo <path to project folder>
disable
```

If you haven't enabled the Windows Subsystem for Linux, you can enter the following line in a PowerShell window *running as Administrator* then reboot when asked:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-
Windows-Subsystem-Linux
```

Version control systems

Introduction

Godot aims to be VCS-friendly and generate mostly readable and mergeable files.

Version control plugins

Godot also supports the use of version control systems in the editor itself. However, version control in the editor requires a plugin for the specific VCS you're using.

As of July 2023, there is only a Git plugin available, but the community may create additional VCS plugins.

Official Git plugin

Using Git from inside the editor is supported with an official plugin. You can find the latest releases on GitHub [https://github.com/godotengine/godot-git-plugin/releases].

Documentation on how to use the Git plugin can be found on its wiki [https://github.com/godotengine/godot-git-plugin/wiki].

Files to exclude from VCS

Note

This lists files and folders that should be ignored from version control in Godot 4.1 and later.

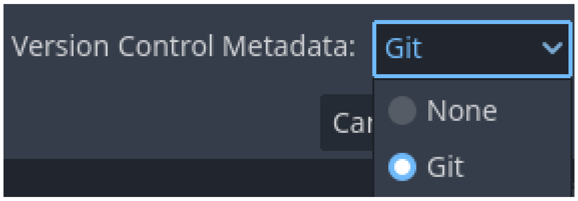
The list of files of folders that should be ignored from version control in Godot 3.x and Godot 4.0 is **entirely** different. This is important, as Godot 3.x and 4.0 may store sensitive credentials in `export_presets.cfg` (unlike Godot 4.1 and later).

If you are using Godot 3, check the `3.5` version of this documentation page instead.

There are some files and folders Godot automatically creates when opening a project in the editor for the first time. To avoid bloating your version control repository with generated data, you should add them to your VCS ignore:

- `.godot/`: This folder stores various project cache data.
- `*.translation`: These files are binary imported translations generated from CSV files.

You can make the Godot project manager generate version control metadata for you automatically when creating a project. When choosing the **Git** option, this creates `.gitignore` and `.gitattributes` files in the project root:



Creating version control metadata in the project manager's **New Project** dialog

In existing projects, select the **Project** menu at the top of the editor, then choose **Version Control > Generate Version**

Control Metadata. This creates the same files as if the operation was performed in the project manager.

Working with Git on Windows

Most Git for Windows clients are configured with the `core.autocrlf` set to `true`. This can lead to files unnecessarily being marked as modified by Git due to their line endings being converted from LF to CRLF automatically.

It is better to set this option as:

```
git config --global core.autocrlf input
```

Creating version control metadata using the project manager or editor will automatically enforce LF line endings using the `.gitattributes` file. In this case, you don't need to change your Git configuration.

Troubleshooting

This page lists common issues encountered when using Godot and possible solutions.

See also

See Using the Web editor for caveats specific to the Web version of the Godot editor.

The editor runs slowly and uses all my CPU and GPU resources, making my computer noisy

This is a known issue, especially on macOS since most Macs have Retina displays. Due to Retina displays' higher pixel density, everything has to be rendered at a higher resolution. This increases the load on the GPU and decreases perceived performance.

There are several ways to improve performance and battery life:

- In 3D, click the **Perspective** button in the top left corner and enable **Half Resolution**. The 3D viewport will now be rendered at half resolution, which can be up to 4 times faster.
- Open the Editor Settings and increase the value of **Low Processor Mode Sleep (µsec)** to `33000` (30 FPS). This value determines the amount of *microseconds* between frames to render. Higher values will make the editor feel less reactive but will help decrease CPU and GPU usage significantly.

- If you have a node that causes the editor to redraw continuously (such as particles), hide it and show it using a script in the `_ready()` method. This way, it will be hidden in the editor but will still be visible in the running project.

The editor stutters and flickers on my variable refresh rate monitor (G-Sync/FreeSync)

This is a known issue [https://github.com/godotengine/godot/issues/38219]. Variable refresh rate monitors need to adjust their gamma curves continuously to emit a consistent amount of light over time. This can cause flicker to appear in dark areas of the image when the refresh rate varies a lot, which occurs as the Godot editor only redraws when necessary.

There are several workarounds for this:

- Enable **Interface > Editor > Update Continuously** in the Editor Settings. Keep in mind this will increase power usage and heat/noise emissions since the editor will now be rendering constantly, even if nothing has changed on screen. To alleviate this, you can increase **Low Processor Mode Sleep (µsec)** to `33000` (30 FPS) in the Editor Settings. This value determines the amount of *microseconds* between frames to render. Higher values will make the editor feel less reactive but will help decrease CPU and GPU usage significantly.
- Alternatively, disable variable refresh rate on your monitor or in the graphics driver.
- VRR flicker can be reduced on some displays using the **VRR Control** or **Fine Tune Dark Areas** options in your monitor's OSD. These options may increase input lag or result in crushed blacks.

- If using an OLED display, use the **Black (OLED)** editor theme preset in the Editor Settings. This hides VRR flicker thanks to OLED's perfect black levels.

The editor or project takes a very long time to start

When using one of the Vulkan-based renderers (Forward+ or Forward Mobile), the first startup is expected to be relatively long. This is because shaders need to be compiled before they can be cached. Shaders also need to be cached again after updating Godot, after updating graphics drivers or after switching graphics cards.

If the issue persists after the first startup, this is a known bug [https://github.com/godotengine/godot/issues/20566] on Windows when you have specific USB peripherals connected. In particular, Corsair's iCUE software seems to cause this bug. Try updating your USB peripherals' drivers to their latest version. If the bug persists, you need to disconnect the specific peripheral before opening the editor. You can then connect the peripheral again.

Firewall software such as Portmaster may also cause the debug port to be blocked. This causes the project to take a long time to start, while being unable to use debugging features in the editor (such as viewing `print()` output). You can work this around by changing the debug port used by the project in the Editor Settings (**Network > Debug > Remote Port**). The default is `6007`; try another value that is greater than `1024`, such as `7007`.

On Windows, when loading the project for the first time after the PC is turned on, Windows Defender will cause the filesystem cache validation on project startup to take significantly longer. This is especially noticeable in projects with a large number of files. Consinder adding the project folder to the list of exclusions by going to Virus & threat protection > Virus & threat protection settings > Add or remove exclusions.

The Godot editor appears frozen after clicking the system console

When running Godot on Windows with the system console enabled, you can accidentally enable *selection mode* by clicking inside the command window. This Windows-specific behavior pauses the application to let you select text inside the system console. Godot cannot override this system-specific behavior.

To solve this, select the system console window and press Enter to leave selection mode.

The Godot editor's macOS dock icon gets duplicated every time it is manually moved

If you open the Godot editor and manually change the position of the dock icon, then restart the editor, you will get a duplicate dock icon all the way to the right of the dock.

This is due to a design limitation of the macOS dock. The only known way to resolve this would be to merge the project manager and editor into a single process, which means the project manager would no longer spawn a separate process when starting the editor. While using a single process instance would bring several benefits, it isn't planned to be done in the near future due to the complexity of the task.

To avoid this issue, keep the Godot editor's dock icon at its default location as created by macOS.

Some text such as "NO DC" appears in the top-left corner of the Project Manager and editor window

This is caused by the NVIDIA graphics driver injecting an overlay to display information.

To disable this overlay on Windows, restore your graphics driver settings to the default values in the NVIDIA Control Panel.

To disable this overlay on Linux, open `nvidia-settings`, go to **X Screen 0 > OpenGL Settings** then uncheck **Enable Graphics API Visual Indicator**.

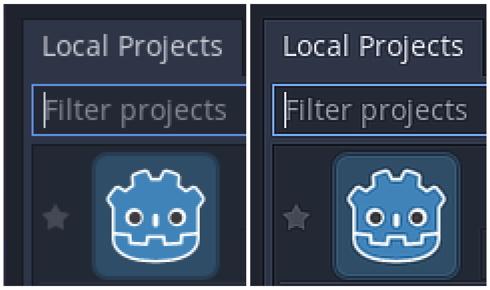
A microphone or "refresh" icon appears in the bottom-right corner of the Project Manager and editor window

This is caused by the NVIDIA graphics driver injecting an overlay to display instant replay information on ShadowPlay recording. This overlay can only be seen on Windows, as Linux does not have support for ShadowPlay.

To disable this overlay, press `Alt + Z` (default shortcut for the NVIDIA overlay) and disable **Settings > HUD Layout > Status Indicator** in the NVIDIA overlay.

Alternatively, you can install the *new NVIDIA app <https://www.nvidia.com/en-us/software/nvidia-app/>* which replaces GeForce Experience and does not suffer from this issue. Unlike GeForce Experience, the NVIDIA app draws the replay indicator in the corner of the screen as opposed to the corner of each window.

The editor or project appears overly sharp or blurry



Correct appearance (left), oversharpened appearance due to graphics driver sharpening (right)

If the editor or project appears overly sharp, this is likely due to image sharpening being forced on all Vulkan or OpenGL applications by your graphics driver. You can disable this behavior in the graphics driver's control panel:

- **NVIDIA (Windows):** Open the start menu and choose **NVIDIA Control Panel**. Open the **Manage 3D settings** tab on the left. In the list in the middle, scroll to **Image Sharpening** and set it to **Sharpening Off**.
- **AMD (Windows):** Open the start menu and choose **AMD Software**. Click the settings "cog" icon in the top-right corner. Go to the **Graphics** tab then disable **Radeon Image Sharpening**.

If the editor or project appears overly blurry, this is likely due to FXAA being forced on all Vulkan or OpenGL applications by your graphics driver.

- **NVIDIA (Windows):** Open the start menu and choose **NVIDIA Control Panel**. Open the **Manage 3D settings** tab on the left. In the list in the middle, scroll to **Fast Approximate Antialiasing** and set it to **Application Controlled**.
- **NVIDIA (Linux):** Open the applications menu and choose **NVIDIA X Server Settings**. Select to **Antialiasing Settings** on the left, then uncheck **Enable FXAA**.
- **AMD (Windows):** Open the start menu and choose **AMD Software**. Click the settings "cog" icon in the top-right corner. Go to the **Graphics** tab, scroll to the bottom and click **Advanced** to unfold its settings. Disable **Morphological Anti-Aliasing**.

Third-party vendor-independent utilities such as vkBasalt may also force sharpening or FXAA on all Vulkan applications. You may want to check their configuration as well.

After changing options in the graphics driver or third-party utilities, restart Godot to make the changes effective.

If you still wish to force sharpening or FXAA on other applications, it's recommended to do so on a per-application basis using the application profiles system provided by graphics drivers' control panels.

The editor or project appears to have washed out colors

On Windows, this is usually caused by incorrect OS or monitor settings, as Godot currently does not support HDR *output* (even though it may internally render in HDR).

As most displays are not designed to display SDR content in HDR mode [https://tftcentral.co.uk/articles/heres-why-you-should-only-enable-hdr-mode-on-your-pc-when-you-are-viewing-hdr-content], it is recommended to disable HDR in the Windows settings when not running applications that use HDR output. On Windows 11, this can be done by pressing `Windows + Alt + B` (this shortcut is part of the Xbox Game Bar app). To toggle HDR automatically based on applications currently running, you can use AutoActions [https://github.com/Codectory/AutoActions].

If you insist on leaving HDR enabled, it is possible to somewhat improve the result by ensuring the display is configured to use HGIG tonemapping (as opposed to DTM), then using the Windows HDR calibration app [https://support.microsoft.com/en-us/windows/calibrate-your-hdr-display-using-the-windows-hdr-calibration-app-f30f4809-3369-43e4-9b02-9eabebd23f19]. It is also strongly recommended to use Windows 11 instead of Windows 10 when using HDR. The end result will still likely be inferior to disabling HDR on the display, though.

Support for HDR *output* is planned in a future release.

The editor/project freezes or displays glitched visuals after resuming the PC from suspend

This is a known issue on Linux with NVIDIA graphics when using the proprietary driver. There is no definitive fix yet, as suspend on Linux + NVIDIA is often buggy when OpenGL or Vulkan is involved. The Compatibility rendering method (which uses OpenGL) is generally less prone to suspend-related issues compared to the Forward+ and Forward Mobile rendering methods (which use Vulkan).

The NVIDIA driver offers an *experimental* option to preserve video memory after suspend [https://wiki.archlinux.org/title/NVIDIA/Tips_and_tricks#Preserve_video_memory_after_suspend] which may resolve this issue. This option has been reported to work better with more recent NVIDIA driver versions.

To avoid losing work, save scenes in the editor before putting the PC to sleep.

The project works when run from the editor, but fails to load some files when running from an exported copy

This is usually caused by forgetting to specify a filter for non-resource files in the Export dialog. By default, Godot will only include actual *resources* into the PCK file. Some files commonly used, such as JSON files, are not considered resources. For example, if you load `test.json` in the exported project, you need to specify `*.json` in the non-resource export filter. See Resource options for more information.

Also, note that files and folders whose names begin with a period will never be included in the exported project. This is done to prevent version control folders like `.git` from being included in the exported PCK file.

On Windows, this can also be due to case sensitivity issues. If you reference a resource in your script with a different case than on the filesystem, loading will fail once you export the project. This is because the virtual PCK filesystem is case-sensitive, while Windows's filesystem is case-insensitive by default.

Editor introduction

In this section, we cover the Godot editor in general, from its interface to using it with the command line.

Editor's interface

The following pages explain how to use the various windows, workspaces, and docks that make up the Godot editor. We cover some specific editors' interface in other sections where appropriate. For example, the animation editor.

- Using the Project Manager
- The Inspector
- Project Settings
- Default editor shortcuts
- Customizing the interface

Android editor

Godot offers a native port of the editor running entirely on Android devices. The Android port can be downloaded from the Android Downloads page [https://godotengine.org/download/android/]. While we strive for feature parity with the Desktop version of the editor, the Android port has a certain amount of caveats you should be aware of.

- Using the Android editor

Web editor

Godot offers an HTML5 version of the editor running entirely in your browser. No download is required to use it, but it has a certain amount of caveats you should be aware of.

- Using the Web editor

Advanced features

The articles below focus on advanced features useful for experienced developers, such as calling Godot from the command line and using an external text editor like Visual Studio Code or Emacs.

- Command line tutorial
- Using an external text editor

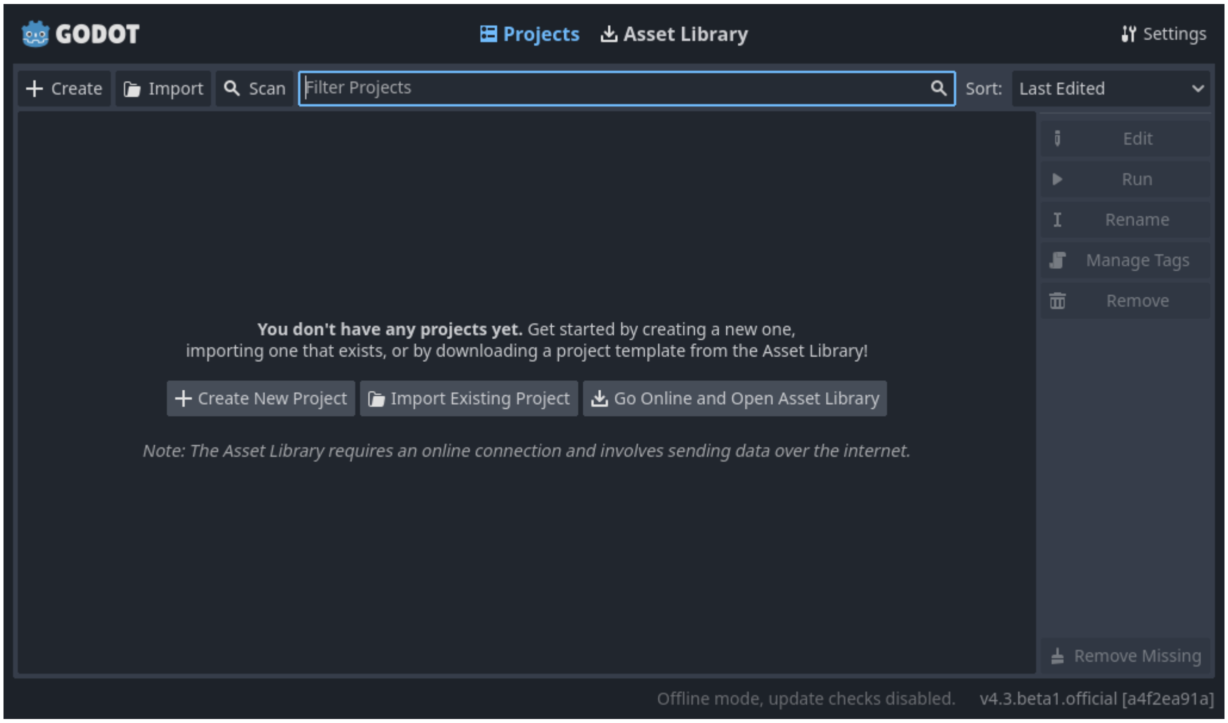
Managing editor features

Godot allows you to remove features from the editor. This may be useful if you're an educator trying to ease students into the editor slowly, or if you're working on a project that's only 2D or only 3D and don't want to see what you don't need.

- Managing editor features

Using the Project Manager

When you launch Godot, the first window you see is the Project Manager. It lets you create, remove, import, or play game projects:

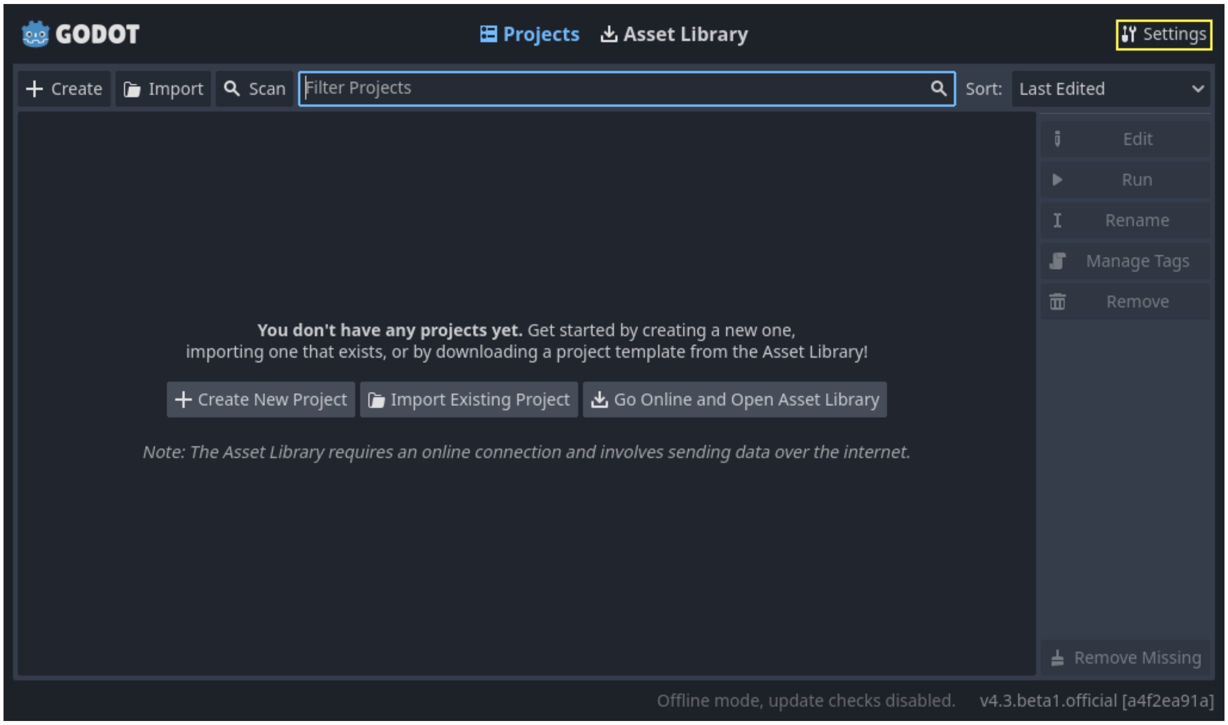

GODOT
Projects
Asset Library
Settings
Create
Import
Scan
Filter Projects
Sort:
Last Edited
Edit
Run
Rename
Manage Tags
Remove
Create New Project
Import Existing Project
Go Online and Open Asset Library
Remove Missing

You don't have any projects yet. Get started by creating a new one, importing one that exists, or by downloading a project template from the Asset Library!

Note: The Asset Library requires an online connection and involves sending data over the internet.

Offline mode, update checks disabled. v4.3.beta1.official [a4f2ea91a]

To change the editors language click on the **Settings** Button in the top right corner:


GODOT
Projects
Asset Library
Settings
Create
Import
Scan
Filter Projects
Sort:
Last Edited
Edit
Run
Rename
Manage Tags
Remove
Create New Project
Import Existing Project
Go Online and Open Asset Library
Remove Missing

You don't have any projects yet. Get started by creating a new one, importing one that exists, or by downloading a project template from the Asset Library!

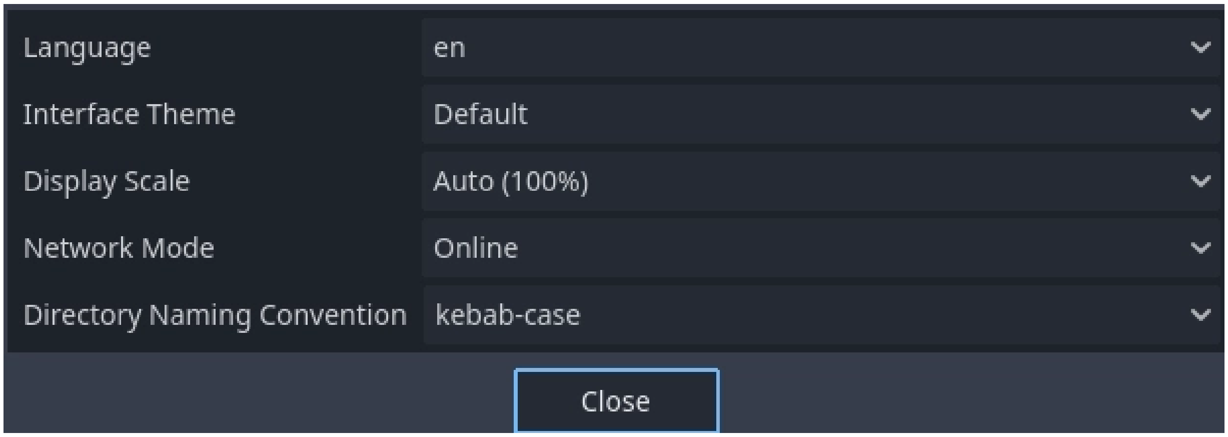
Note: The Asset Library requires an online connection and involves sending data over the internet.

Offline mode, update checks disabled. v4.3.beta1.official [a4f2ea91a]

In Project Manager Settings, you can change the interface **language** from the language dropdown menu, which is the system default language by default.

You can also change the **theme** of the editor, the **display scale** for different interface element sizes, and the availability of online functionality using **network mode**. If network mode is online, Godot will also check and inform you about new versions of Godot.

The **directory naming convention** can also be changed to replace spaces according to the chosen format when creating folders automatically.

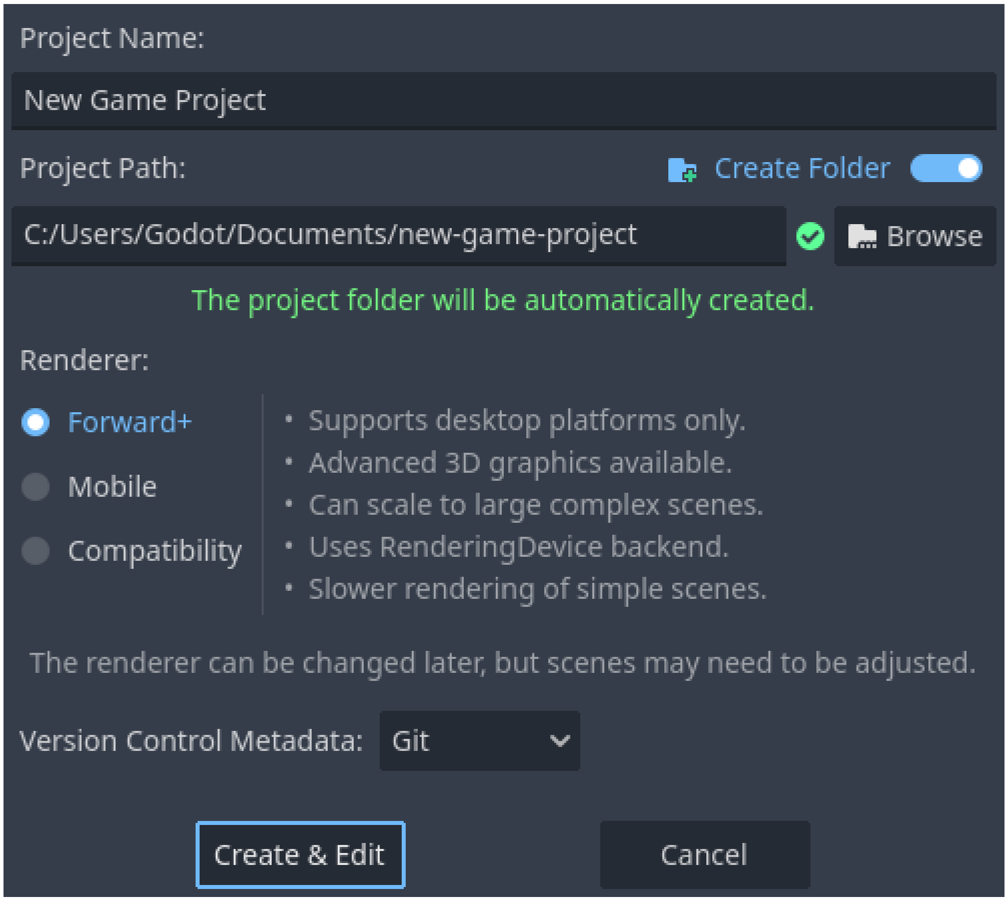


Language	en
Interface Theme	Default
Display Scale	Auto (100%)
Network Mode	Online
Directory Naming Convention	kebab-case

Creating and importing projects

To create a new project:

1. Click the **Create** button on the top-left of the window.
2. Give the project a name, choose an empty folder on your computer to save the files. Alternatively, you can enable **Create Folder** option to automatically create a new sub-folder with the project name, following the directory naming convention set in the settings. An empty folder will show a green tick on the right.
3. Select one of the rendering backends (this can also be changed later).
4. Click the **Create & Edit** button to create the project folder and open it in the editor.



Project Name:
New Game Project

Project Path:
C:/Users/Godot/Documents/new-game-project

The project folder will be automatically created.

Renderer:

● Forward+
○ Mobile
○ Compatibility

- Supports desktop platforms only.
- Advanced 3D graphics available.
- Can scale to large complex scenes.
- Uses RenderingDevice backend.
- Slower rendering of simple scenes.

The renderer can be changed later, but scenes may need to be adjusted.

Version Control Metadata: Git

Note

You can optionally choose a version control system. Currently, only git [https://git-scm.com] is supported and it needs the Godot Git Plugin to be installed, either manually or using the Asset Library. To learn more about the Godot Git Plugin, see its wiki [https://github.com/godotengine/godot-git-plugin/wiki].

Using the file browser

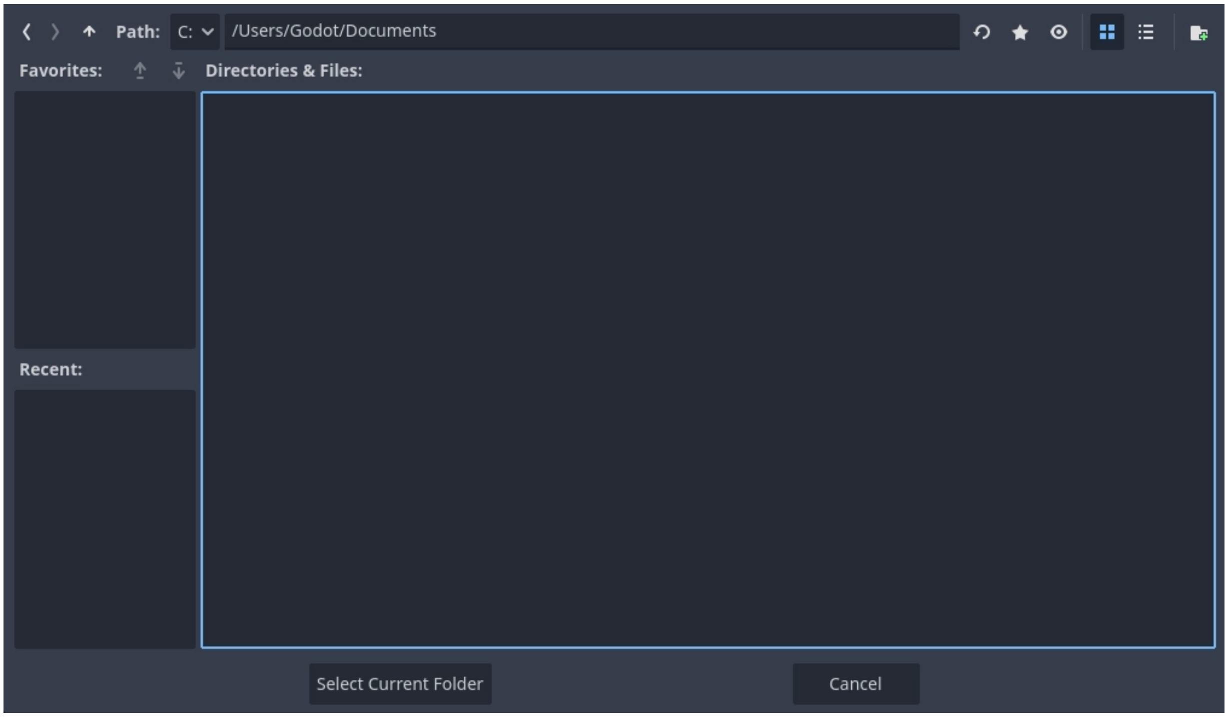
Click the **Browse** button to open Godot's file browser. You can pick a location or type the folder's path in the **Path** field, after choosing a drive.

Left of the path field on the top row contains arrows to navigate backward and forward through the last visited locations. The up arrow navigates to parent folder. On the right side of the path field, there are buttons to refresh the current folder's contents, favorite/unfavorite the current folder, and show/hide hidden folders.

Next, the buttons to switch the display type of the folders and files between grid view and list view are seen.

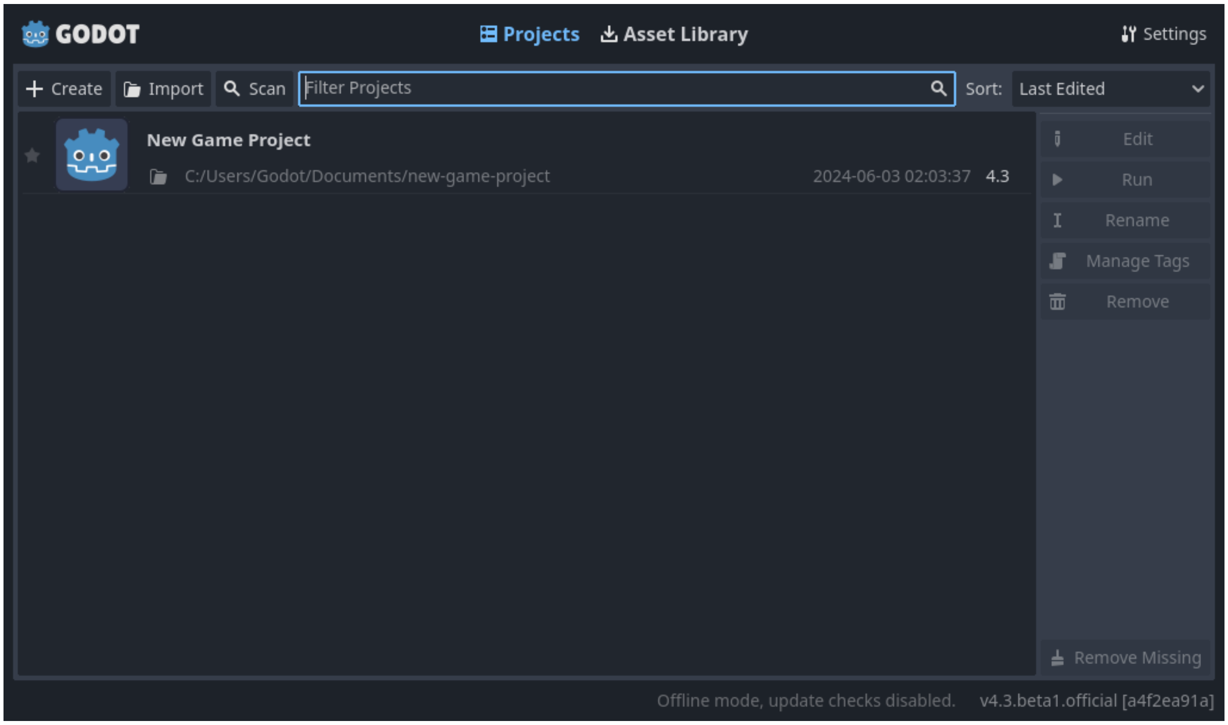
The last button on the right will create a new folder.

Favorited folders will be displayed on the left side under the **Favorites** section. You can sort the favorites using the up and down buttons in this section. Last chosen folders will be listed under the **Recent** list.

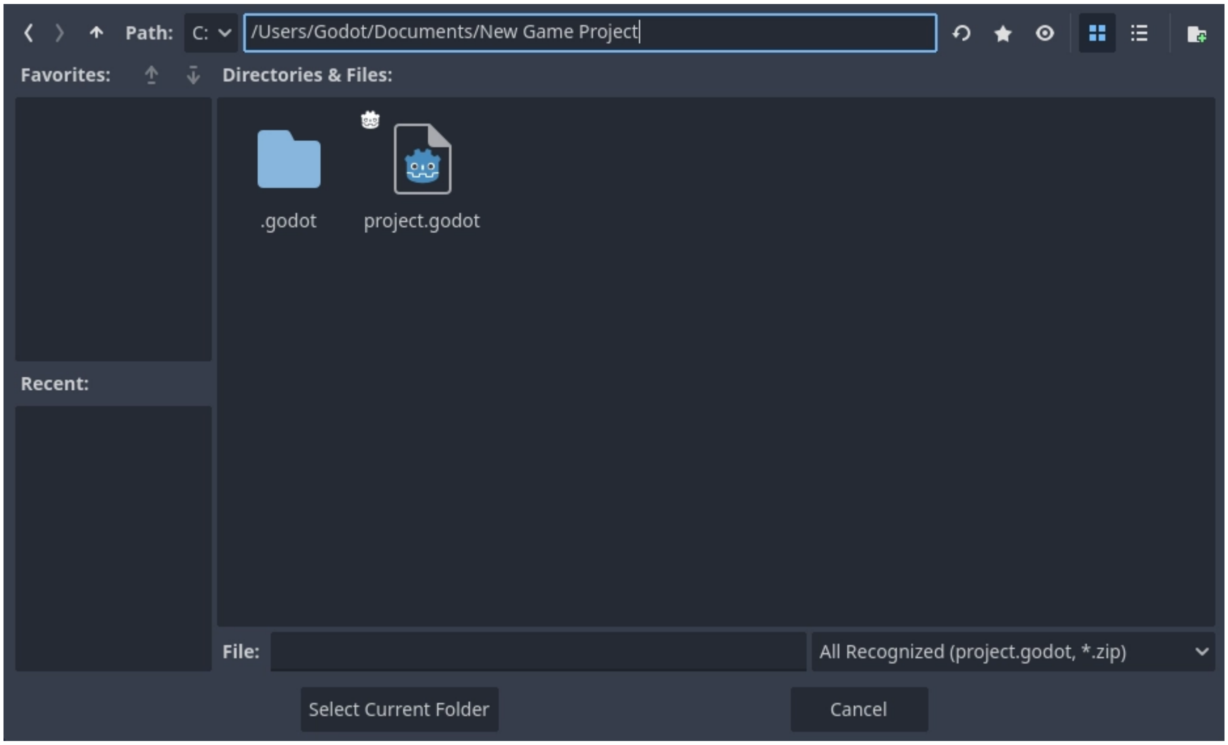

Path:
C:
/Users/Godot/Documents
Favorites:
Directories & Files:
Recent:
Select Current Folder
Cancel

Opening and importing projects

The next time you open the Project Manager, you'll see your new project in the list. Double click on it to open it in the editor.


GODOT
Projects
Asset Library
Settings
Create
Import
Scan
Filter Projects
Sort:
Last Edited
New Game Project
C:/Users/Godot/Documents/new-game-project
2024-06-03 02:03:37
4.3
Edit
Run
Rename
Manage Tags
Remove
Remove Missing
Offline mode, update checks disabled.
v4.3.beta1.official [a4f2ea91a]

You can similarly import existing projects using the **Import** button. Locate the folder that contains the project or the **project.godot** file to import and edit it.


Path: C: /Users/Godot/Documents/New Game Project

.godot

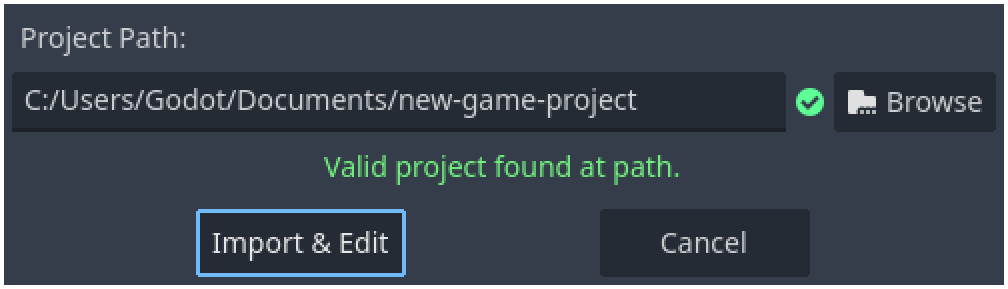
project.godot

All Recognized (project.godot, *.zip)

Favorites:
Directories & Files:
Recent:
File:
Select Current Folder
Cancel

Alternatively, it is possible to choose a zip file to be automatically extracted by Godot.

When the folder path is correct, you'll see a green checkmark.


Project Path: C:/Users/Godot/Documents/new-game-project ✓

Valid project found at path.

Browse
Import & Edit
Cancel

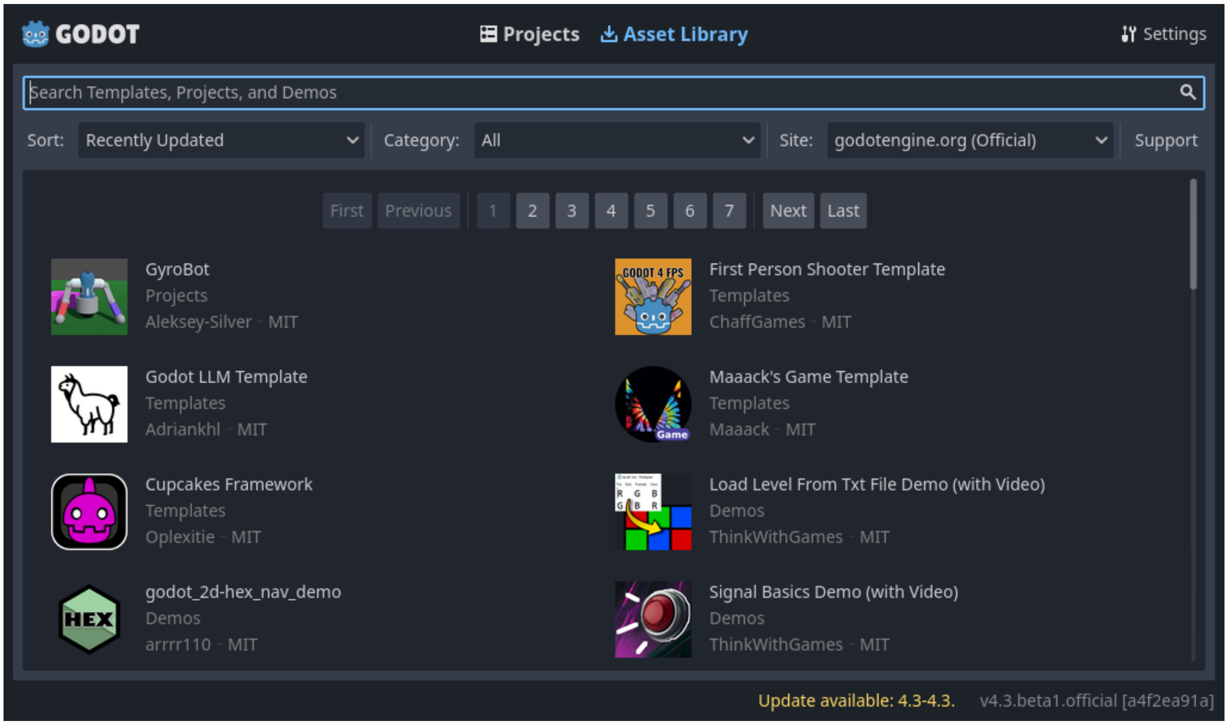
Downloading demos and templates

From the **Asset Library** tab you can download open source project templates and demos from the [Asset Library]() to help you get started faster.

The first time you open this tab you'll notice that it's asking you to go online. For privacy reasons the project manager, and Godot editor, can't access the internet by default. To enable accessing the internet click the **Go Online** button. This will also allow project manager to notify you about updates. If you wish to turn this off in the future go into project manager settings and change **Network Mode** to "Offline"

Now that Godot is connected to the internet you can download a demo or template, to do this:

1. Click on its title.
2. On the page that opens, click the download button.
3. Once it finished downloading, click install and choose where you want to save the project.



Sort: Recently Updated

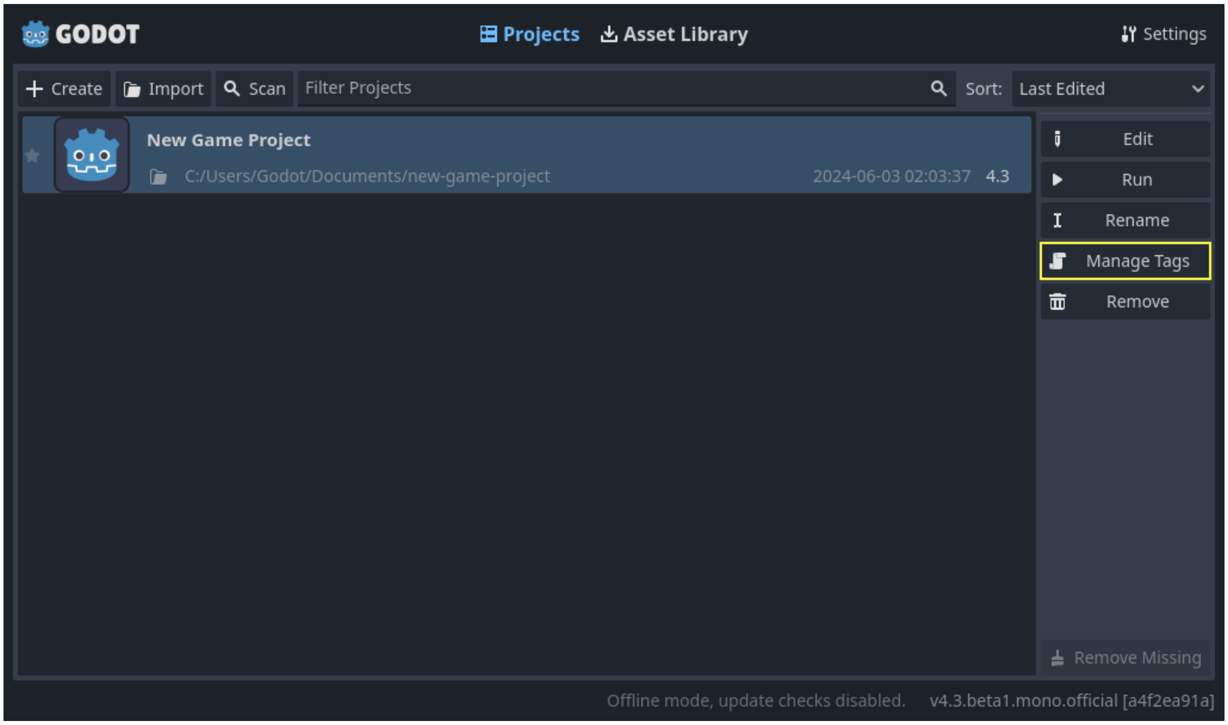
Category: All

Site: godotengine.org (Official)

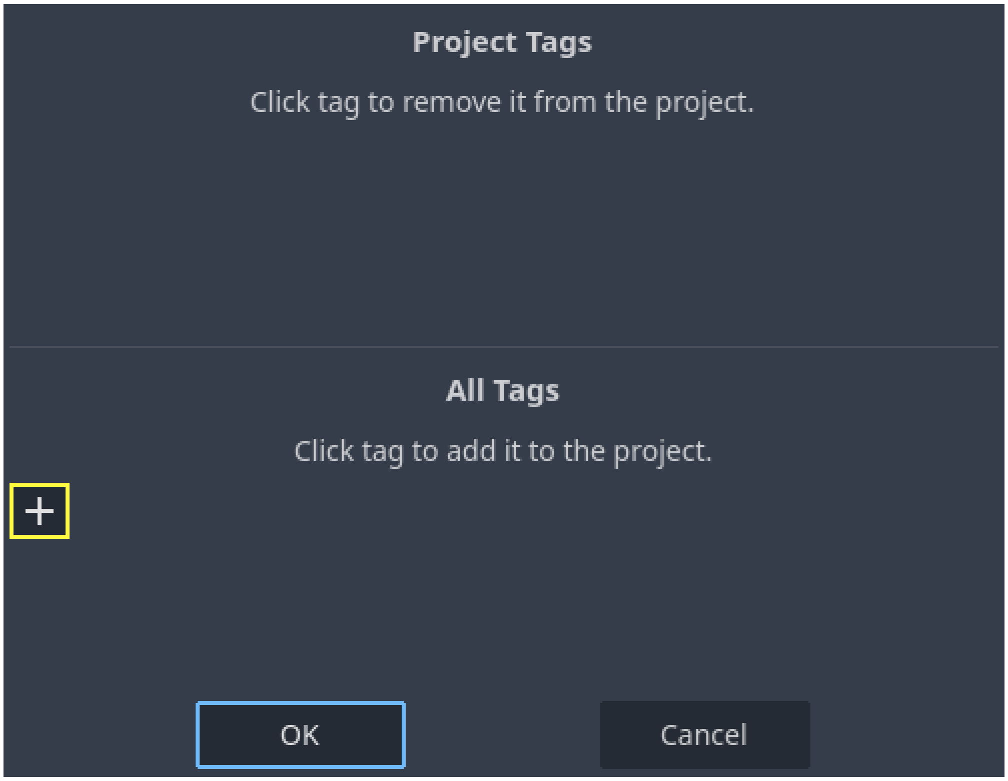
GODOT
Projects
Asset Library
Settings
Search Templates, Projects, and Demos
Support
First
Previous
1
2
3
4
5
6
7
Next
Last
GyroBot
Projects
Aleksey-Silver · MIT
First Person Shooter Template
Templates
ChaffGames · MIT
Godot LLM Template
Templates
Adriankhl · MIT
Maaack's Game Template
Templates
Maaack · MIT
Cupcakes Framework
Templates
Oplexitie · MIT
Load Level From Txt File Demo (with Video)
Demos
ThinkWithGames · MIT
godot_2d-hex_nav_demo
Demos
arrrr110 · MIT
Signal Basics Demo (with Video)
Demos
ThinkWithGames · MIT
Update available: 4.3-4.3.
v4.3.beta1.official [a4f2ea91a]

Managing projects with tags

For users with a lot of projects on one PC it can be a lot to keep track of. To aid in this Godot allows you to create project tags. To add a tag to a project click on the project in the project manager, then click on the **Manage Tags** button


GODOT
Projects
Asset Library
Settings
Create
Import
Scan
Filter Projects
Sort:
Last Edited
New Game Project
C:/Users/Godot/Documents/new-game-project
2024-06-03 02:03:37
4.3
Edit
Run
Rename
Manage Tags
Remove
Remove Missing
Offline mode, update checks disabled.
v4.3.beta1.mono.official [a4f2ea91a]

This will open up the manage project tags window. To add a tag click the plus button.



Project Tags
Click tag to remove it from the project.
All Tags
Click tag to add it to the project.
+
OK
Cancel

Type out the tag name, and click **OK**. Your project will now have a tag added to it. These tags can be used for any other project in your project manager.

To show projects with a specific tag only, you can click on the tags or write `tag:` and type the tag you would like to search for in the filter bar. To limit the results using multiple tags, you can click on another tag or add `tag:` after a space and type another tag in the filter bar.

In addition, tags will stay with projects. So if you tag your project, send it to another machine, and import it into the project manager you will see the tags you created.

To remove a tag from your project manager it must be removed from all the projects it's used by. Once that's done close the project manager, open it up again, and the tag should be gone.

The Inspector

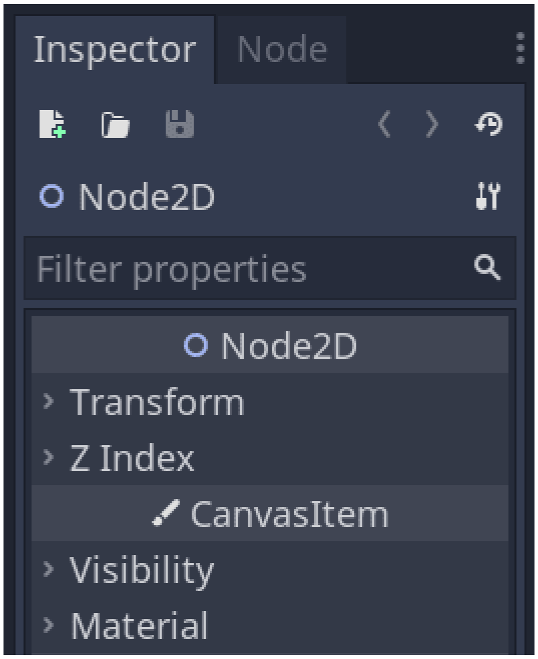
This page explains how the Inspector dock works in-depth. You will learn how to edit properties, fold and unfold areas, use the search bar, and more.

Warning

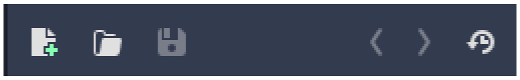
This page is a work-in-progress.

Overview of the interface

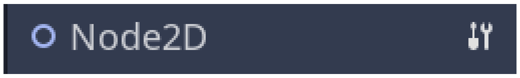
Let's start by looking at the dock's main parts.


Inspector
Node
Node2D
Filter properties
Node2D
Transform
Z Index
CanvasItem
Visibility
Material

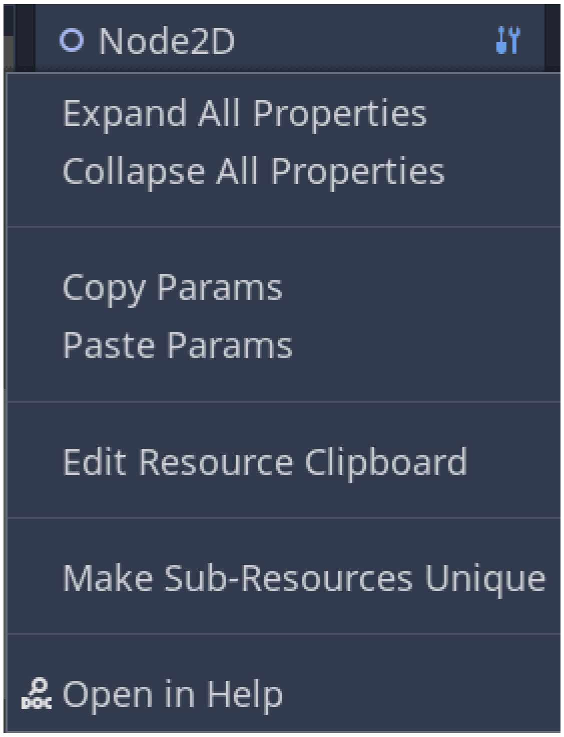
At the top are the file and navigation buttons.



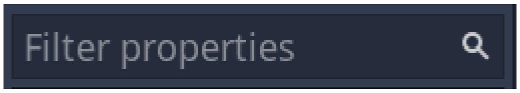
Below it, you can find the selected node's name, its type, and the tools menu on the right side.



If you click the tool menu icon, a drop-down menu offers some view and edit options.


Node2D
Expand All Properties
Collapse All Properties
Copy Params
Paste Params
Edit Resource Clipboard
Make Sub-Resources Unique
Open in Help

Then comes the search bar. Type anything in it to filter displayed properties. Delete the text to clear the search.


Filter properties

Project Settings

This page explains how to use the Project Settings window. If you would like to access and modify project settings via code, see ProjectSettings.

Godot stores the project settings in a project.godot file, a plain text file in INI format. There are dozens of settings you can change to control a project's execution. To simplify this process, Godot provides a project settings dialog, which acts as a front-end to editing a project.godot file.

To access that dialog, select Project -> Project Settings.

Once the window opens, let's select a main scene. Locate the *Application/Run/Main Scene* property and click on it to select 'hello.tscn'.

The project settings dialog provides a lot of options that can be saved to a project.godot file and shows their default values. If you change a value, a tick appears to the left of its name. This means that the property will be saved in the project.godot file and remembered.

Default editor shortcuts

Many of Godot Editor functions can be executed with keyboard shortcuts. This page lists functions which have associated shortcuts by default, but many others are available for customization in editor settings as well. To change keys associated with these and other actions navigate to `Editor -> Editor Settings -> Shortcuts`.

While some actions are universal, a lot of shortcuts are specific to individual tools. For this reason it is possible for some key combinations to be assigned to more than one function. The correct action will be performed depending on the context.

Note

While Windows and Linux builds of the editor share most of the default settings, some shortcuts may differ for macOS version. This is done for better integration of the editor into macOS ecosystem. Users fluent with standard shortcuts on that OS should find Godot Editor's default key mapping intuitive.

General Editor Actions

Action name	Windows, Linux	macOS	Editor setting
Open 2D Editor	`Ctrl + F1`	`Opt + 1`	`editor/editor_2d`
Open 3D Editor	`Ctrl + F2`	`Opt + 2`	`editor/editor_3d`
Open Script Editor	`Ctrl + F3`	`Opt + 3`	`editor/editor_script`
Search Help	`F1`	`Opt + Space`	`editor/editor_help`
Distraction Free Mode	`Ctrl + Shift + F11`	`Cmd + Ctrl + D`	`editor/distraction_free_mode`
Next tab	`Ctrl + Tab`	`Cmd + Tab`	`editor/next_tab`
Previous tab	`Ctrl + Shift + Tab`	`Cmd + Shift + Tab`	`editor/prev_tab`

Action name	Windows, Linux	macOS	Editor setting
Filter Files	`Ctrl + Alt + P`	`Opt + Cmd + P`	`editor/filter_files`
Open Scene	`Ctrl + O`	`Cmd + O`	`editor/open_scene`
Close Scene	`Ctrl + Shift + W`	`Cmd + Shift + W`	`editor/close_scene`
Reopen Closed Scene	`Ctrl + Shift + T`	`Cmd + Shift + T`	`editor/reopen_closed_scene`
Save Scene	`Ctrl + S`	`Cmd + S`	`editor/save_scene`
Save Scene As	`Ctrl + Shift + S`	`Cmd + Shift + S`	`editor/save_scene_as`
Save All Scenes	`Ctrl + Shift + Alt + S`	`Cmd + Shift + Opt + S`	`editor/save_all_scenes`
Quick Open	`Shift + Alt + O`	`Shift + Opt + O`	`editor/quick_open`
Quick Open Scene	`Ctrl + Shift + O`	`Cmd + Shift + O`	`editor/quick_open_scene`
Quick Open Script	`Ctrl + Alt + O`	`Opt + Cmd + O`	`editor/quick_open_script`
Undo	`Ctrl + Z`	`Cmd + Z`	`editor/undo`
Redo	`Ctrl + Shift + Z`	`Cmd + Shift + Z`	`editor/redo`
Quit	`Ctrl + Q`	`Cmd + Q`	`editor/file_quit`
Quit to Project List	`Ctrl + Shift + Q`	`Shift + Opt + Q`	`editor/quit_to_project_list`

Action name	Windows, Linux	macOS	Editor setting
Take Screenshot	`Ctrl + F12`	`Cmd + F12`	`editor/take_screenshot`
Toggle Fullscreen	`Shift + F11`	`Cmd + Ctrl + F`	`editor/fullscreen_mode`
Play	`F5`	`Cmd + B`	`editor/play`
Pause Scene	`F7`	`Cmd + Ctrl + Y`	`editor/pause_scene`
Stop	`F8`	`Cmd + .`	`editor/stop`
Play Scene	`F6`	`Cmd + R`	`editor/play_scene`
Play Custom Scene	`Ctrl + Shift + F5`	`Cmd + Shift + R`	`editor/play_custom_scene`
Expand Bottom Panel	`Shift + F12`	`Shift + F12`	`editor/bottom_panel_expand`

2D / Canvas Item Editor

Action name	Windows, Linux	macOS	Editor setting
Zoom In	`Ctrl + =`	`Cmd + =`	`canvas_item_editor/zoom_plus`
Zoom Out	`Ctrl + -`	`Cmd + -`	`canvas_item_editor/zoom_minus`
Zoom Reset	`Ctrl + 0`	`Cmd + 0`	`canvas_item_editor/zoom_reset`
Pan View	`Space`	`Space`	`canvas_item_editor/pan_view`

Action name	Windows, Linux	macOS	Editor setting
Select Mode	`Q`	`Q`	`canvas_item_editor/select_mode`
Move Mode	`W`	`W`	`canvas_item_editor/move_mode`
Rotate Mode	`E`	`E`	`canvas_item_editor/rotate_mode`
Scale Mode	`S`	`S`	`canvas_item_editor/scale_mode`
Ruler Mode	`R`	`R`	`canvas_item_editor/ruler_mode`
Use Smart Snap	`Shift + S`	`Shift + S`	`canvas_item_editor/use_smart_snap`
Use Grid Snap	`Shift + G`	`Shift + G`	`canvas_item_editor/use_grid_snap`
Multiply grid step by 2	`Num *`	`Num *`	`canvas_item_editor/multiply_grid_step`
Divide grid step by 2	`Num /`	`Num /`	`canvas_item_editor/divide_grid_step`
Always Show Grid	`G`	`G`	`canvas_item_editor/show_grid`
Show Helpers	`H`	`H`	`canvas_item_editor/show_helpers`
Show Guides	`Y`	`Y`	`canvas_item_editor/show_guides`

Action name	Windows, Linux	macOS	Editor setting
Center Selection	`F`	`F`	`canvas_item_editor/center_selection`
Frame Selection	`Shift + F`	`Shift + F`	`canvas_item_editor/frame_selection`
Preview Canvas Scale	`Ctrl + Shift + P`	`Cmd + Shift + P`	`canvas_item_editor/preview_canvas_scale`
Insert Key	`Ins`	`Ins`	`canvas_item_editor/anim_insert_key`
Insert Key (Existing Tracks)	`Ctrl + Ins`	`Cmd + Ins`	`canvas_item_editor/anim_insert_key_existing_tracks`
Make Custom Bones from Nodes	`Ctrl + Shift + B`	`Cmd + Shift + B`	`canvas_item_editor/skeleton_make_bones`
Clear Pose	`Shift + K`	`Shift + K`	`canvas_item_editor/anim_clear_pose`

3D / Spatial Editor

Action name	Windows, Linux	macOS	Editor setting
Toggle Freelook	`Shift + F`	`Shift + F`	`spatial_editor/freelook_toggle`
Freelook Left	`A`	`A`	`spatial_editor/freelook_left`
Freelook Right	`D`	`D`	`spatial_editor/freelook_right`

Action name	**Windows, Linux**	**macOS**	**Editor setting**
Freelook Forward	`W`	`W`	`spatial_editor/freelook_forward`
Freelook Backwards	`S`	`S`	`spatial_editor/freelook_backwards`
Freelook Up	`E`	`E`	`spatial_editor/freelook_up`
Freelook Down	`Q`	`Q`	`spatial_editor/freelook_down`
Freelook Speed Modifier	`Shift`	`Shift`	`spatial_editor/freelook_speed_modifier`
Freelook Slow Modifier	`Alt`	`Opt`	`spatial_editor/freelook_slow_modifier`
Select Mode	`Q`	`Q`	`spatial_editor/tool_select`
Move Mode	`W`	`W`	`spatial_editor/tool_move`
Rotate Mode	`E`	`E`	`spatial_editor/tool_rotate`
Scale Mode	`R`	`R`	`spatial_editor/tool_scale`
Use Local Space	`T`	`T`	`spatial_editor/local_coords`
Use Snap	`Y`	`Y`	`spatial_editor/snap`
Snap Object to Floor	`PgDown`	`PgDown`	`spatial_editor/snap_to_floor`
Top View	`Num 7`	`Num 7`	`spatial_editor/top_view`
Bottom View	`Alt + Num 7`	`Opt + Num 7`	`spatial_editor/bottom_view`

Action name	Windows, Linux	macOS	Editor setting
Front View	`Num 1`	`Num 1`	`spatial_editor/front_view`
Rear View	`Alt + Num 1`	`Opt + Num 1`	`spatial_editor/rear_view`
Right View	`Num 3`	`Num 3`	`spatial_editor/right_view`
Left View	`Alt + Num 3`	`Opt + Num 3`	`spatial_editor/left_view`
Switch Perspective/Orthogonal View	`Num 5`	`Num 5`	`spatial_editor/switch_perspective_orthogonal`
Insert Animation Key	`K`	`K`	`spatial_editor/insert_anim_key`
Focus Origin	`O`	`O`	`spatial_editor/focus_origin`
Focus Selection	`F`	`F`	`spatial_editor/focus_selection`
Align Transform with View	`Ctrl + Alt + M`	`Opt + Cmd + M`	`spatial_editor/align_transform_with_view`
Align Rotation with View	`Ctrl + Alt + F`	`Opt + Cmd + F`	`spatial_editor/align_rotation_with_view`
1 Viewport	`Ctrl + 1`	`Cmd + 1`	`spatial_editor/1_viewport`
2 Viewports	`Ctrl + 2`	`Cmd + 2`	`spatial_editor/2_viewports`
2 Viewports (Alt)	`Ctrl + Alt + 2`	`Opt + Cmd + 2`	`spatial_editor/2_viewports_alt`

Action name	Windows, Linux	macOS	Editor setting
3 Viewports	`Ctrl + 3`	`Cmd + 3`	`spatial_editor/3_viewports`
3 Viewports (Alt)	`Ctrl + Alt + 3`	`Opt + Cmd + 3`	`spatial_editor/3_viewports_alt`
4 Viewports	`Ctrl + 4`	`Cmd + 4`	`spatial_editor/4_viewports`

Text Editor

Action name	Windows, Linux	macOS	Editor setting
Cut	`Ctrl + X`	`Cmd + X`	`script_text_editor/cut`
Copy	`Ctrl + C`	`Cmd + C`	`script_text_editor/copy`
Paste	`Ctrl + V`	`Cmd + V`	`script_text_editor/paste`
Select All	`Ctrl + A`	`Cmd + A`	`script_text_editor/select_all`
Find	`Ctrl + F`	`Cmd + F`	`script_text_editor/find`
Find Next	`F3`	`Cmd + G`	`script_text_editor/find_next`
Find Previous	`Shift + F3`	`Cmd + Shift + G`	`script_text_editor/find_previous`
Find in Files	`Ctrl + Shift + F`	`Cmd + Shift + F`	`script_text_editor/find_in_files`
Replace	`Ctrl + R`	`Opt + Cmd + F`	`script_text_editor/replace`

Action name	Windows, Linux	macOS	Editor setting
Replace in Files	`Ctrl + Shift + R`	`Cmd + Shift + R`	`script_text_editor/replace_in_files`
Undo	`Ctrl + Z`	`Cmd + Z`	`script_text_editor/undo`
Redo	`Ctrl + Y`	`Cmd + Y`	`script_text_editor/redo`
Move Up	`Alt + Up Arrow`	`Opt + Up Arrow`	`script_text_editor/move_up`
Move Down	`Alt + Down Arrow`	`Opt + Down Arrow`	`script_text_editor/move_down`
Delete Line	`Ctrl + Shift + K`	`Cmd + Shift + K`	`script_text_editor/delete_line`
Toggle Comment	`Ctrl + K`	`Cmd + K`	`script_text_editor/toggle_comment`
Fold/Unfold Line	`Alt + F`	`Ctrl + Cmd + F`	`script_text_editor/toggle_fold_line`
Duplicate Lines	`Ctrl + Alt + Down Arrow`	`Cmd + Shift + Down Arrow`	`script_text_editor/duplicate_lines`
Duplicate Selection	`Ctrl + Shift + D`	`Cmd + Shift + C`	`script_text_editor/duplicate_selection`
Complete Symbol	`Ctrl + Space`	`Ctrl + Space`	`script_text_editor/complete_symbol`
Evaluate Selection	`Ctrl + Shift + E`	`Cmd + Shift + E`	`script_text_editor/evaluate_selection`

Action name	Windows, Linux	macOS	Editor setting
Trim Trailing Whitespace	`Ctrl + Alt + T`	`Opt + Cmd + T`	`script_text_editor/trim_trailing_whitespace`
Uppercase	`Shift + F4`	`Shift + F4`	`script_text_editor/convert_to_uppercase`
Lowercase	`Shift + F5`	`Shift + F5`	`script_text_editor/convert_to_lowercase`
Capitalize	`Shift + F6`	`Shift + F6`	`script_text_editor/capitalize`
Convert Indent to Spaces	`Ctrl + Shift + Y`	`Cmd + Shift + Y`	`script_text_editor/convert_indent_to_spaces`
Convert Indent to Tabs	`Ctrl + Shift + I`	`Cmd + Shift + I`	`script_text_editor/convert_indent_to_tabs`
Auto Indent	`Ctrl + I`	`Cmd + I`	`script_text_editor/auto_indent`
Toggle Bookmark	`Ctrl + Alt + B`	`Opt + Cmd + B`	`script_text_editor/toggle_bookmark`
Go to Next Bookmark	`Ctrl + B`	`Cmd + B`	`script_text_editor/goto_next_bookmark`
Go to Previous Bookmark	`Ctrl + Shift + B`	`Cmd + Shift + B`	`script_text_editor/goto_previous_bookmark`
Go to Function	`Ctrl + Alt + F`	`Ctrl + Cmd + J`	`script_text_editor/goto_function`
Go to Line	`Ctrl + L`	`Cmd + L`	`script_text_editor/goto_line`

Action name	Windows, Linux	macOS	Editor setting
Toggle Breakpoint	`F9`	`Cmd + Shift + B`	`script_text_editor/toggle_breakpoint`
Remove All Breakpoints	`Ctrl + Shift + F9`	`Cmd + Shift + F9`	`script_text_editor/remove_all_breakpoints`
Go to Next Breakpoint	`Ctrl + .`	`Cmd + .`	`script_text_editor/goto_next_breakpoint`
Go to Previous Breakpoint	`Ctrl + ,`	`Cmd + ,`	`script_text_editor/goto_previous_breakpoint`
Contextual Help	`Alt + F1`	`Opt + Shift + Space`	`script_text_editor/contextual_help`

Script Editor

Action name	Windows, Linux	macOS	Editor setting
Find	`Ctrl + F`	`Cmd + F`	`script_editor/find`
Find Next	`F3`	`F3`	`script_editor/find_next`
Find Previous	`Shift + F3`	`Shift + F3`	`script_editor/find_previous`
Find in Files	`Ctrl + Shift + F`	`Cmd + Shift + F`	`script_editor/find_in_files`
Move Up	`Shift + Alt + Up Arrow`	`Shift + Opt + Up Arrow`	`script_editor/window_move_up`

Action name	**Windows, Linux**	**macOS**	**Editor setting**
Move Down	`Shift + Alt + Down Arrow`	`Shift + Opt + Down Arrow`	`script_editor/window_move_down`
Next Script	`Ctrl + Shift + .`	`Cmd + Shift + .`	`script_editor/next_script`
Previous Script	`Ctrl + Shift + ,`	`Cmd + Shift + ,`	`script_editor/prev_script`
Reopen Closed Script	`Ctrl + Shift + T`	`Cmd + Shift + T`	`script_editor/reopen_closed_script`
Save	`Ctrl + Alt + S`	`Opt + Cmd + S`	`script_editor/save`
Save All	`Ctrl + Shift + Alt + S`	`Cmd + Shift + Opt + S`	`script_editor/save_all`
Soft Reload Script	`Ctrl + Shift + R`	`Cmd + Shift + R`	`script_editor/reload_script_soft`
History Previous	`Alt + Left Arrow`	`Opt + Left Arrow`	`script_editor/history_previous`
History Next	`Alt + Right Arrow`	`Opt + Right Arrow`	`script_editor/history_next`
Close	`Ctrl + W`	`Cmd + W`	`script_editor/close_file`
Run	`Ctrl + Shift + X`	`Cmd + Shift + X`	`script_editor/run_file`
Toggle Scripts Panel	`Ctrl + \`	`Cmd + \`	`script_editor/toggle_scripts_panel`
Zoom In	`Ctrl + =`	`Cmd + =`	`script_editor/zoom_in`

Action name	Windows, Linux	macOS	Editor setting
Zoom Out	`Ctrl + -`	`Cmd + -`	`script_editor/zoom_out`
Reset Zoom	`Ctrl + 0`	`Cmd + 0`	`script_editor/reset_zoom`

Editor Output

Action name	Windows, Linux	macOS	Editor setting
Copy Selection	`Ctrl + C`	`Cmd + C`	`editor/copy_output`
Clear Output	`Ctrl + Shift + K`	`Cmd + Shift + K`	`editor/clear_output`

Debugger

Action name	Windows, Linux	macOS	Editor setting
Step Into	`F11`	`F11`	`debugger/step_into`
Step Over	`F10`	`F10`	`debugger/step_over`
Continue	`F12`	`F12`	`debugger/continue`

File Dialog

Action name	Windows, Linux	macOS	Editor setting
Go Back	`Alt + Left Arrow`	`Opt + Left Arrow`	`file_dialog/go_back`
Go Forward	`Alt + Right Arrow`	`Opt + Right Arrow`	`file_dialog/go_forward`

Action name	Windows, Linux	macOS	Editor setting
Go Up	`Alt + Up Arrow`	`Opt + Up Arrow`	`file_dialog/go_up`
Refresh	`F5`	`F5`	`file_dialog/refresh`
Toggle Hidden Files	`Ctrl + H`	`Cmd + H`	`file_dialog/toggle_hidden_files`
Toggle Favorite	`Alt + F`	`Opt + F`	`file_dialog/toggle_favorite`
Toggle Mode	`Alt + V`	`Opt + V`	`file_dialog/toggle_mode`
Create Folder	`Ctrl + N`	`Cmd + N`	`file_dialog/create_folder`
Delete	`Del`	`Cmd + BkSp`	`file_dialog/delete`
Focus Path	`Ctrl + D`	`Cmd + D`	`file_dialog/focus_path`
Move Favorite Up	`Ctrl + Up Arrow`	`Cmd + Up Arrow`	`file_dialog/move_favorite_up`
Move Favorite Down	`Ctrl + Down Arrow`	`Cmd + Down Arrow`	`file_dialog/move_favorite_down`

FileSystem Dock

Action name	Windows, Linux	macOS	Editor setting
Copy Path	`Ctrl + C`	`Cmd + C`	`filesystem_dock/copy_path`
Duplicate	`Ctrl + D`	`Cmd + D`	`filesystem_dock/duplicate`

Action name	Windows, Linux	macOS	Editor setting
Delete	`Del`	`Cmd + BkSp`	`filesystem_dock/delete`

Scene Tree Dock

Action name	Windows, Linux	macOS	Editor setting
Add Child Node	`Ctrl + A`	`Cmd + A`	`scene_tree/add_child_node`
Batch Rename	`Ctrl + F2`	`Cmd + F2`	`scene_tree/batch_rename`
Copy Node Path	`Ctrl + Shift + C`	`Cmd + Shift + C`	`scene_tree/copy_node_path`
Delete	`Del`	`Cmd + BkSp`	`scene_tree/delete`
Force Delete	`Shift + Del`	`Shift + Del`	`scene_tree/delete_no_confirm`
Duplicate	`Ctrl + D`	`Cmd + D`	`scene_tree/duplicate`
Move Up	`Ctrl + Up Arrow`	`Cmd + Up Arrow`	`scene_tree/move_up`
Move Down	`Ctrl + Down Arrow`	`Cmd + Down Arrow`	`scene_tree/move_down`

Animation Track Editor

Action name	Windows, Linux	macOS	Editor setting
Duplicate Selection	`Ctrl + D`	`Cmd + D`	`animation_editor/duplicate_selection`
Duplicate Transposed	`Ctrl + Shift + D`	`Cmd + Shift + D`	`animation_editor/duplicate_selection_transposed`
Delete Selection	`Del`	`Cmd + BkSp`	`animation_editor/delete_selection`
Go to Next Step	`Ctrl + Right Arrow`	`Cmd + Right Arrow`	`animation_editor/goto_next_step`
Go to Previous Step	`Ctrl + Left Arrow`	`Cmd + Left Arrow`	`animation_editor/goto_prev_step`

Tile Map Editor

Action name	Windows, Linux	macOS	Editor setting
Select	`S`	`S`	`tiles_editor/selection_tool`
Cut Selection	`Ctrl + X`	`Cmd + X`	`tiles_editor/cut`
Copy Selection	`Ctrl + C`	`Cmd + C`	`tiles_editor/copy`
Paste Selection	`Ctrl + V`	`Cmd + V`	`tiles_editor/paste`
Delete Selection	`Del`	`Cmd + BkSp`	`tiles_editor/delete`
Cancel	`Esc`	`Esc`	`tiles_editor/cancel`

Action name	Windows, Linux	macOS	Editor setting
Paint	`D`	`D`	`tiles_editor/paint_tool`
Line	`L`	`L`	`tiles_editor/line_tool`
Rect	`R`	`R`	`tiles_editor/rect_tool`
Bucket	`B`	`B`	`tiles_editor/bucket_tool`
Picker	`P`	`P`	`tiles_editor/picker`
Eraser	`E`	`E`	`tiles_editor/eraser`
Flip Horizontally	`C`	`C`	`tiles_editor/flip_tile_horizontal`
Flip Vertically	`V`	`V`	`tiles_editor/flip_tile_vertical`
Rotate Left	`Z`	`Z`	`tiles_editor/rotate_tile_left`
Rotate Right	`X`	`X`	`tiles_editor/rotate_tile_right`

Tileset Editor

Action name	Windows, Linux	macOS	Editor setting
Next Coordinate	`PgDown`	`PgDown`	`tileset_editor/next_shape`
Previous Coordinate	`PgUp`	`PgUp`	`tileset_editor/previous_shape`
Region Mode	`1`	`1`	`tileset_editor/editmode_region`

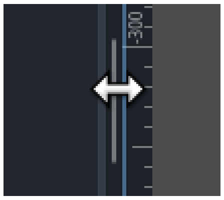
Action name	Windows, Linux	macOS	Editor setting
Collision Mode	`2`	`2`	`tileset_editor/editmode_collision`
Occlusion Mode	`3`	`3`	`tileset_editor/editmode_occlusion`
Navigation Mode	`4`	`4`	`tileset_editor/editmode_navigation`
Bitmask Mode	`5`	`5`	`tileset_editor/editmode_bitmask`
Priority Mode	`6`	`6`	`tileset_editor/editmode_priority`
Icon Mode	`7`	`7`	`tileset_editor/editmode_icon`
Z Index Mode	`8`	`8`	`tileset_editor/editmode_z_index`

Customizing the interface

Godot's interface lives in a single window by default. Since Godot 4.0, you can split several elements to separate windows to better make use of multi-monitor setups.

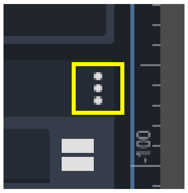
Moving and resizing docks

Click and drag on the edge of any dock or panel to resize it horizontally or vertically:



Resizing a dock in the editor

Click the "3 vertical dots" icon at the top of any dock to change its location, or split it to a separate window by choosing **Make Floating** in the submenu that appears:



Moving a dock in the editor

To move a floating dock back to the editor window, close the dock window using the × button in the top-right corner of the window (or in the top-left corner on macOS). Alternatively, you can press `Alt + F4` while the split window is focused.

Splitting the script or shader editor to its own window

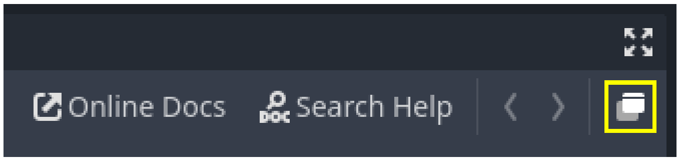
Note

This feature is only available on platforms that support spawning multiple windows: Windows, macOS and Linux.

This feature is also not available if **Single Window Mode** is enabled in the Editor Settings.

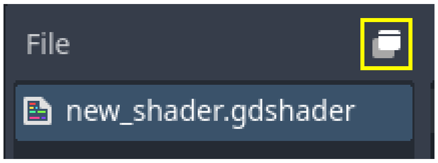
Since Godot 4.1, you can split the script or shader editor to its own window.

To split the script editor to its own window, click the corresponding button in the top-right corner of the script editor:



Splitting the script editor to its own window

To split the shader editor to its own window, click the corresponding button in the top-right corner of the script editor:



Splitting the shader editor to its own window

To go back to the previous state (with the script/shader editor embedded in the editor window), close the split window using the ×

button in the top-right corner of the window (or in the top-left corner on macOS). Alternatively, you can press `Alt + F4` while the split window is focused.

Customizing editor layouts

You may want to save and load a dock configuration depending on the kind of task you're working on. For instance, when working on animating a character, it may be more convenient to have docks laid out in a different fashion compared to when you're designing a level.

For this purpose, Godot provides a way to save and restore editor layouts. Before saving a layout, make changes to the docks you'd like to save. The following changes are persisted to the saved layout:

- Moving a dock.
- Resizing a dock.
- Making a dock floating.
- Changing a floating dock's position or size.
- FileSystem dock properties: split mode, display mode, sorting order, file list display mode, selected paths and unfolded paths.

Note

Splitting the script or shader editor to its own window is *not* persisted as part of a layout.

After making changes, open the **Editor** menu at the top of the editor then choose **Editor Layouts > Save**. Enter a name for the layout, then click **Save**. If you've already saved an editor layout, you can choose to override an existing layout using the list.

After making changes, open the **Editor** menu at the top of the editor then choose **Editor Layouts**. In the dropdown list, you will see a list of saved editor layouts, plus **Default** which is a hardcoded editor layout that can't be removed. The default layout matches a fresh Godot installation with no changes made to the docks' position and size, and no floating docks.

You can remove a layout using the **Delete** option in the **Editor Layouts** dropdown.

Tip

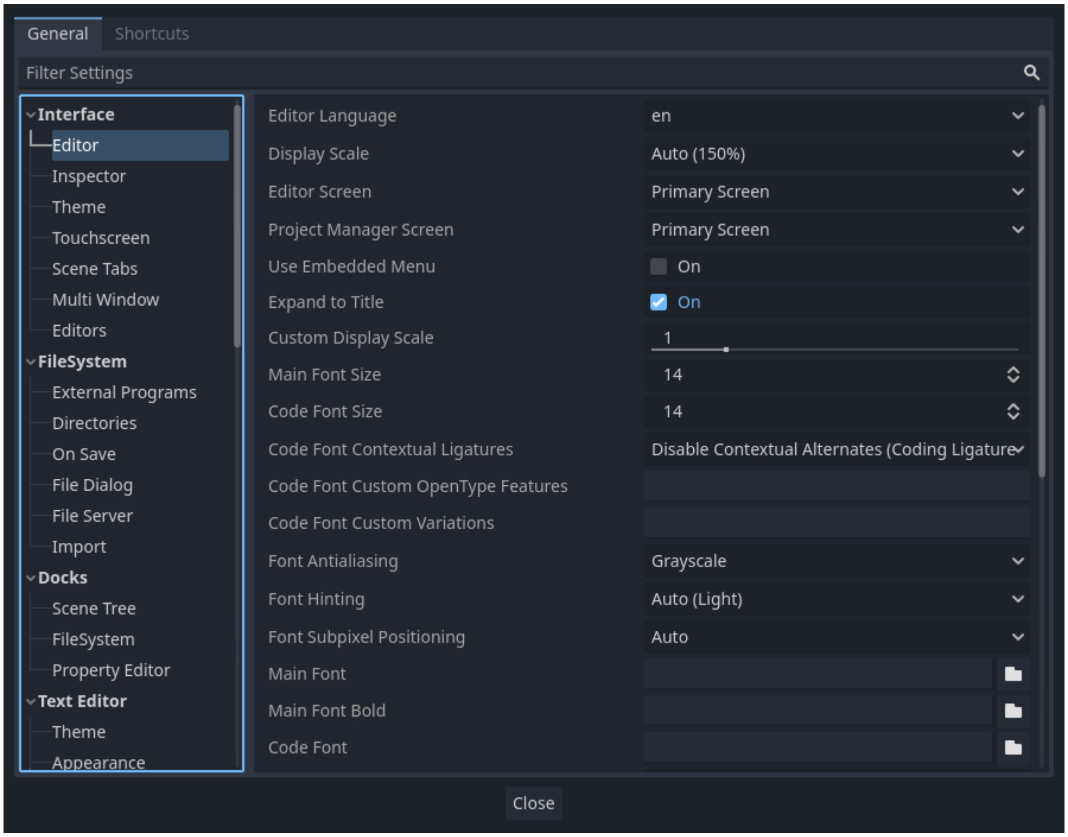
If you name the saved layout `Default` (case-sensitive), the default editor layout will be overwritten. Note that the `Default` does not appear in the list of layouts to overwrite until you overwrite it once, but you can still write its name manually.

You can go back to the standard default layout by removing the `Default` layout after overriding it. (This option does not appear if you haven't overridden the default layout yet.)

Editor layouts are saved to a file named `editor_layouts.cfg` in the configuration path of the Editor data paths.

Customizing editor settings

In the **Editor** menu at the top of the editor, you can find an **Editor Settings** option. This opens a window similar to the Project Settings, but with settings used by the editor. These settings are shared across all projects and are not saved in the project files.


General
Shortcuts
Filter Settings
Interface
Editor
Inspector
Theme
Touchscreen
Scene Tabs
Multi Window
Editors
FileSystem
External Programs
Directories
On Save
File Dialog
File Server
Import
Docks
Scene Tree
FileSystem
Property Editor
Text Editor
Theme
Appearance
Close

Editor Language	en
Display Scale	Auto (150%)
Editor Screen	Primary Screen
Project Manager Screen	Primary Screen
Use Embedded Menu	☐ On
Expand to Title	☑ On
Custom Display Scale	1
Main Font Size	14
Code Font Size	14
Code Font Contextual Ligatures	Disable Contextual Alternates (Coding Ligature
Code Font Custom OpenType Features	
Code Font Custom Variations	
Font Antialiasing	Grayscale
Font Hinting	Auto (Light)
Font Subpixel Positioning	Auto
Main Font	
Main Font Bold	
Code Font	

The Editor Settings window

Some commonly changed settings are:

- **Interface > Editor > Editor Language:** Controls the language the editor displays in. To make English tutorials easier to follow, you may want to change this to English so that menu names are identical to names referred to by tutorials. The language can also be changed in the top-right corner of the project manager.
- **Interface > Editor > Display Scale:** Controls how large UI elements display on screen. The default **Auto** setting finds a suitable value based on your display's DPI and resolution. Due to engine limitations, it only takes the display-provided scaling factor on macOS, not on Windows or Linux.
- **Interface > Editor > Single Window Mode:** If enabled, this forces the editor to use a single window. This disables certain features such as splitting the script/shaders editor to their own

window. Single-window mode can be more stable, especially on Linux when using Wayland.

- **Interface > Theme > Preset:** The editor theme preset to use. The **Light** theme preset may be easier to read if you're outdoors or in a room with sunlight. The **Black (OLED)** preset can reduce power consumption on OLED displays, which are increasingly common in laptops and phones/tablets.
- **FileSystem > Directories > Autoscan Project Path:** This can be set to a folder path that will be automatically scanned for projects in the project manager every time it starts.
- **FileSystem > Directories > Default Project Path:** Controls the default location where new projects are created in the project manager.
- **Editors > 3D > Emulate Numpad:** This allows using the top row 0-9 keys in the 3D editor as their equivalent numpad keys. It's recommended to enable this option if you don't have a number pad on your keyboard.
- **Editors > 3D > Emulate 3 Button Mouse:** This allows using the pan, zoom and orbit modifiers in the 3D editor even when not holding down any mouse button. It's recommended to enable this option if you're using a trackpad.

See the EditorSettings class reference for a complete description of most editor settings. You can also hover an editor setting's name with the mouse in the Editor Settings to show its description.

Using the Android editor

In 2023, we added [https://godotengine.org/article/android_godot_editor_play_store_beta_release/] an Android port of the editor [https://godotengine.org/download/android/] that can be used to work on new or existing projects on Android devices.

Note

The Android editor is in early access, while we continue to refine the experience, and bring it up to parity with the Desktop version of the editor. See Required Permissions below.

Android devices support

The Android editor requires devices running Android 5 Lollipop or higher, with at least OpenGL 3 support. This includes (not exhaustive):

- Android tablets, foldables and large phones
- Android-powered netbooks
- Chromebooks supporting Android apps

Required Permissions

The Android editor requires the All files access permission [https://developer.android.com/training/data-storage/manage-all-files#all-files-access]. The permission allows the editor to create / import / read project files from any file locations on the device. Without the permission, the editor is still functional, but has limited access to the device's files and directories.

Limitations & known issues

Here are the known limitations and issues of the Android editor:

- No C#/Mono support
- No support for external script editors
- While available, the *Vulkan Forward+* renderer is not recommended due to severe performance issues
- No support for building and exporting an Android APK binary. As a workaround, you can generate and export a Godot PCK or ZIP file [https://docs.godotengine.org/en/stable/tutorials/export/exporting_projects.html#pck-versus-zip-pack-file-formats]
- No support for building and exporting binaries for other platforms
- UX not optimized for Android phones form-factor
- Android Go devices [https://developer.android.com/guide/topics/androidgo] lacks the *All files access* permission required for device read/write access. As a workaround, when using an Android Go device, it's recommended to create new projects only in the Android *Documents* or *Downloads* directories.
- The editor doesn't properly resume when *Don't keep activities* is enabled in the *Developer Options*

See also

See the list of open issues on GitHub related to the Android editor [https://github.com/godotengine/godot/issues?q=is%3Aopen+is%3Aissue+label%3Aplatform%3Aandroid+label%3Atopic%3Aeditor] for a list of known bugs.

Using the Web editor

Since Godot 3.3, there is a Web editor [https://editor.godotengine.org/] you can use to work on new or existing projects.

Note

The web editor is in a preliminary stage. While its feature set may be sufficient for educational purposes, it is currently **not recommended for production work**. See Limitations below.

Browser support

The Web editor requires support for WebAssembly's SharedArrayBuffer. This is in turn required to support threading in the browser. The following desktop browsers support WebAssembly threading and can therefore run the web editor:

- Chrome 68 or later
- Firefox 79 or later
- Edge 79 or later

Opera and Safari are not supported yet. Safari may work in the future once proper threading support is added.

Mobile browsers are currently not supported.

The web editor only supports the Compatibility rendering method, as there is no stable way to run Vulkan applications on the web yet.

Note

If you use Linux, due to poor Firefox WebGL performance [https://bugzilla.mozilla.org/show_bug.cgi?id=1010527], it's recommended to use a Chromium-based browser instead of Firefox.

Limitations

Due to limitations on the Godot or Web platform side, the following features are currently missing:

- No C#/Mono support.
- No GDExtension support.
- No debugging support. This means GDScript debugging/profiling, live scene editing, the Remote Scene tree dock and other features that rely on the debugger protocol will not work.
- No project exporting. As a workaround, you can download the project source using **Project > Tools > Download Project Source** and export it using a native version of the Godot editor [https://godotengine.org/download].
- The editor won't warn you when closing the tab with unsaved changes.
- No lightmap baking support. You can still use existing lightmaps if they were baked with a native version of the Godot editor (e.g. by importing an existing project).

The following features are unlikely to be supported due to inherent limitations of the Web platform:

- No support for external script editors.
- No support for Android one-click deploy.

See also

See the list of open issues on GitHub related to the web editor [https://github.com/godotengine/godot/issues?q=is%3Aopen+is%3Aissue+label%3Aplatform%3Ahtml5+label%3Atopic%3Aeditor] for a list of known bugs.

Importing a project

To import an existing project, the current process is as follows:

- Specify a ZIP file to preload on the HTML5 filesystem using the **Preload project ZIP** input.
- Run the editor by clicking **Start Godot editor**. The Godot Project Manager should appear after 10-20 seconds. On slower machines or connections, loading may take up to a minute.
- In the dialog that appears at the middle of the window, specify a name for the folder to create then click the **Create Folder** button (it doesn't have to match the ZIP archive's name).
- Click **Install & Edit** and the project will open in the editor.

Attention

It's important to place the project folder somewhere in `/home/web_user/`. If your project folder is placed outside `/home/web_user/`, you will lose your project when closing the editor!

When you follow the steps described above, the project folder will always be located in `/home/web_user/projects`, keeping it safe.

Editing and running a project

Unlike the native version of Godot, the web editor is constrained to a single window. Therefore, it cannot open a new window when running the project. Instead, when you run the project by clicking the Run button or pressing `F5`, it will appear to "replace" the editor window.

The web editor offers an alternative way to deal with the editor and game windows (which are now "tabs"). You can switch between the **Editor** and **Game** tabs using the buttons on the top. You can also close the running game or editor by clicking the **×** button next to those tabs.

Where are my project files?

Due to browser security limitations, the editor will save the project files to the browser's IndexedDB storage. This storage isn't accessible as a regular folder on your machine, but is abstracted away in a database.

You can download the project files as a ZIP archive by using **Project > Tools > Download Project Source**. This can be used to export the project using a native Godot editor [https://godotengine.org/download], since exporting from the web editor isn't supported yet.

In the future, it may be possible to use the HTML5 FileSystem API [https://developer.mozilla.org/en-US/docs/Web/API/FileSystem] to store the project files on the user's filesystem as the native editor would do. However, this isn't implemented yet.

Command line tutorial

Some developers like using the command line extensively. Godot is designed to be friendly to them, so here are the steps for working entirely from the command line. Given the engine relies on almost no external libraries, initialization times are pretty fast, making it suitable for this workflow.

Note

On Windows and Linux, you can run a Godot binary in a terminal by specifying its relative or absolute path.

On macOS, the process is different due to Godot being contained within an `.app` bundle (which is a *folder*, not a file). To run a Godot binary from a terminal on macOS, you have to `cd` to the folder where the Godot application bundle is located, then run `Godot.app/Contents/MacOS/Godot` followed by any command line arguments. If you've renamed the application bundle from `Godot` to another name, make sure to edit this command line accordingly.

Command line reference

Legend

- ☑ Available in editor builds, debug export templates and release export templates.
- 🐞 Available in editor builds and debug export templates only.
- 🛠 Only available in editor builds.

Note that unknown command line arguments have no effect whatsoever. The engine will **not** warn you when using a command line argument that doesn't exist with a given build type.

General options

Command	Description
`-h, --help`	☑ Display the list of command line options.
`--version`	☑ Display the version string.
`-v, --verbose`	☑ Use verbose stdout mode.
`-q, --quiet`	☑ Quiet mode, silences stdout messages. Errors are still displayed.

Run options

Command	Description
`--, ++`	☑ Separator for user-provided arguments. Following arguments are not used by the engine, but can be read from `OS.get_cmdline_user_args()`.
`-e, --editor`	🛠 Start the editor instead of running the scene.
`-p, --project-manager`	🛠 Start the Project Manager, even if a project is auto-detected.

`--debug-server <uri>`	🛠 Start the editor debug server (`<protocol>://<host/IP>[:<port>]`, e.g. `tcp://127.0.0.1:6007`)
`--quit`	☑ Quit after the first iteration.
`--quit-after`	☑ Quit after the given number of iterations. Set to 0 to disable.
`-l, --language <locale>`	☑ Use a specific locale. `<locale>` follows the format `language_Script_COUNTRY_VARIANT` where language is a 2 or 3-letter language code in lowercase and the rest is optional. See Locale codes for more details.
`--path <directory>`	☑ Path to a project (`<directory>` must contain a 'project.godot' file).
`-u, --upwards`	☑ Scan folders upwards for 'project.godot' file.
`--main-pack <file>`	☑ Path to a pack (.pck) file to load.
`--render-thread <mode>`	☑ Render thread mode ('unsafe', 'safe', 'separate'). See Thread Model for more details.
`--remote-fs <address>`	☑ Remote filesystem (`<host/IP>[:<port>]` address).

`--remote-fs-password <password>`	☑ Password for remote filesystem.
`--audio-driver <driver>`	☑ Audio driver. Use `--help` first to display the list of available drivers.
`--display-driver <driver>`	☑ Display driver (and rendering driver). Use `--help` first to display the list of available drivers.
`--rendering-method <renderer>`	☑ Renderer name. Requires driver support.
`--rendering-driver <driver>`	☑ Rendering driver (depends on display driver). Use `--help` first to display the list of available drivers.
`--gpu-index <device_index>`	☑ Use a specific GPU (run with `--verbose` to get available device list).
`--text-driver <driver>`	☑ Text driver (Fonts, BiDi, shaping).
`--tablet-driver`	☑ Pen tablet input driver.

`<driver>`	
`--headless`	☑ Enable headless mode (`--display-driver headless --audio-driver Dummy`). Useful for servers and with `--script`.
`--write-movie <file>`	☑ Run the engine in a way that a movie is written (usually with .avi or .png extension). `--fixed-fps` is forced when enabled, but can be used to change movie FPS. `--disable-vsync` can speed up movie writing but makes interaction more difficult. `--quit-after` can be used to specify the number of frames to write.

Display options

Command	Description
`-f, --fullscreen`	☑ Request fullscreen mode.
`-m, --maximized`	☑ Request a maximized window.
`-w, --windowed`	☑ Request windowed mode.
`-t, --always-on-top`	☑ Request an always-on-top window.
`--resolution <W>x<H>`	☑ Request window resolution.

`--position <X>,<Y>`	☑ Request window position.
`--screen <N>`	☑ Request window screen.
`--single-window`	☑ Use a single window (no separate subwindows).
`--xr-mode <mode>`	☑ Select XR mode ('default', 'off', 'on').

Debug options

Command	Description
`-d, --debug`	☑ Debug (local stdout debugger).
`-b, --breakpoints`	☑ Breakpoint list as source::line comma-separated pairs, no spaces (use `%20` instead).
`--profiling`	☑ Enable profiling in the script debugger.
`--gpu-profile`	☑ Show a GPU profile of the tasks that took the most time during frame rendering.
`--gpu-validation`	☑ Enable graphics API validation layers for debugging.

`--gpu-abort`	🐞 Abort on GPU errors (usually validation layer errors), may help see the problem if your system freezes.
`--remote-debug <uri>`	☑ Remote debug (`<protocol>://<host/IP>[:<port>]`, e.g. `tcp://127.0.0.1:6007`).
`--single-threaded-scene`	☑ Scene tree runs in single-threaded mode. Sub-thread groups are disabled and run on the main thread.
`--debug-collisions`	🐞 Show collision shapes when running the scene.
`--debug-paths`	🐞 Show path lines when running the scene.
`--debug-navigation`	🐞 Show navigation polygons when running the scene.
`--debug-avoidance`	🐞 Show navigation avoidance debug visuals when running the scene.
`--debug-stringnames`	🐞 Print all StringName allocations to stdout when the engine quits.
`--frame-delay <ms>`	☑ Simulate high CPU load (delay each frame by <ms> milliseconds).

`--time-scale <scale>`	☑ Force time scale (higher values are faster, 1.0 is normal speed).
`--disable-vsync`	☑ Forces disabling of vertical synchronization, even if enabled in the project settings. Does not override driver-level V-Sync enforcement.
`--disable-render-loop`	☑ Disable render loop so rendering only occurs when called explicitly from script.
`--disable-crash-handler`	☑ Disable crash handler when supported by the platform code.
`--fixed-fps <fps>`	☑ Force a fixed number of frames per second. This setting disables real-time synchronization.
`--delta-smoothing <enable>`	☑ Enable or disable frame delta smoothing ('enable', 'disable').
`--print-fps`	☑ Print the frames per second to the stdout.

Standalone tools

Command	Description

`-s, --script <script>`	☑ Run a script.
`--check-only`	☑ Only parse for errors and quit (use with `--script`).
`--import`	🛠 Starts the editor, waits for any resources to be imported, and then quits. Implies `--editor` and `--quit`.
`--export-release <preset> <path>`	🛠 Export the project using the given preset and matching release template. The preset name should match one defined in export_presets.cfg. `<path>` should be absolute or relative to the project directory, and include the filename for the binary (e.g. 'builds/game.exe'). The target directory should exist. Implies `--import`.
`--export-debug <preset> <path>`	🛠 Like `--export-release`, but use debug template. Implies `--import`.
`--export-pack <preset> <path>`	🛠 Like `--export-release`, but only export the game pack for the given preset. The `<path>` extension determines whether it will be in PCK or ZIP format. Implies `--import`.
`--convert-3to4 [<max_file_kb>]`	🛠 Convert project from Godot 3.x to Godot 4.x.

`[<max_line_size>]`	
`--validate-conversion-3to4 [<max_file_kb>] [<max_line_size>]`	🛠 Show what elements will be renamed when converting project from Godot 3.x to Godot 4.x.
`--doctool [<path>]`	🛠 Dump the engine API reference to the given `<path>` in XML format, merging if existing files are found.
`--no-docbase`	🛠 Disallow dumping the base types (used with `--doctool`).
`--gdscript-docs <path>`	🛠 Rather than dumping the engine API, generate API reference from the inline documentation in the GDScript files found in <path> (used with `--doctool`).
`--build-solutions`	🛠 Build the scripting solutions (e.g. for C# projects). Implies `--editor` and requires a valid project to edit.
`--dump-gdextension-interface`	🛠 Generate GDExtension header file 'gdnative_interface.h' in the current folder. This file is the base file required to implement a GDExtension.

`--dump-extension-api`	🛠 Generate JSON dump of the Godot API for GDExtension bindings named 'extension_api.json' in the current folder.
`--validate-extension-api <path>`	🛠 Validate an extension API file dumped (with the option above) from a previous version of the engine to ensure API compatibility. If incompatibilities or errors are detected, the return code will be non-zero.
`--benchmark`	🛠 Benchmark the run time and print it to console.
`--benchmark-file <path>`	🛠 Benchmark the run time and save it to a given file in JSON format. The path should be absolute.

Path

It is recommended to place your Godot editor binary in your `PATH` environment variable, so it can be executed easily from any place by typing `godot`. You can do so on Linux by placing the Godot binary in `/usr/local/bin` and making sure it is called `godot` (case-sensitive).

To achieve this on Windows or macOS easily, you can install Godot using Scoop [https://scoop.sh] (on Windows) or Homebrew [https://brew.sh] (on macOS). This will automatically make the copy of Godot installed available in the `PATH`:

Windows

```
# Add "Extras" bucket
scoop bucket add extras

# Standard editor:
scoop install godot

# Editor with C# support (will be available as `godot-mono` in
`PATH`):
scoop install godot-mono
```

macOS

```
# Standard editor:
brew install godot

# Editor with C# support (will be available as `godot-mono` in
`PATH`):
brew install godot-mono
```

Setting the project path

Depending on where your Godot binary is located and what your current working directory is, you may need to set the path to your project for any of the following commands to work correctly.

When running the editor, this can be done by giving the path to the `project.godot` file of your project as either the first argument, like this:

```
godot path_to_your_project/project.godot [other] [commands]
[and] [args]
```

For all commands, this can be done by using the `--path` argument:

```
godot --path path_to_your_project [other] [commands] [and]
[args]
```

For example, the full command for exporting your game (as explained below) might look like this:

```
godot --headless --path path_to_your_project --export-release
my_export_preset_name game.exe
```

When starting from a subdirectory of your project, use the `--upwards` argument for Godot to automatically find the `project.godot` file by recursively searching the parent directories.

For example, running a scene (as explained below) nested in a subdirectory might look like this when your working directory is in the same path:

```
godot --upwards nested_scene.tscn
```

Creating a project

Creating a project from the command line can be done by navigating the shell to the desired place and making a `project.godot` file.

```
mkdir newgame
cd newgame
touch project.godot
```

The project can now be opened with Godot.

Running the editor

Running the editor is done by executing Godot with the `-e` flag. This must be done from within the project directory or by setting the project path as explained above, otherwise the command is ignored and the Project Manager appears.

```
godot -e
```

When passing in the full path to the `project.godot` file, the `-e` flag may be omitted.

If a scene has been created and saved, it can be edited later by running the same code with that scene as argument.

```
godot -e scene.tscn
```

Erasing a scene

Godot is friends with your filesystem and will not create extra metadata files. Use `rm` to erase a scene file. Make sure nothing references that scene. Otherwise, an error will be thrown upon opening the project.

```
rm scene.tscn
```

Running the game

To run the game, execute Godot within the project directory or with the project path as explained above.

```
godot
```

Note that passing in the `project.godot` file will always run the editor instead of running the game.

When a specific scene needs to be tested, pass that scene to the command line.

```
godot scene.tscn
```

Debugging

Catching errors in the command line can be a difficult task because they scroll quickly. For this, a command line debugger is provided by adding `-d`. It works for running either the game or a single scene.

```
godot -d
```

```
godot -d scene.tscn
```

Exporting

Exporting the project from the command line is also supported. This is especially useful for continuous integration setups.

Note

Using the `--headless` command line argument is **required** on platforms that do not have GPU access (such as continuous integration). On platforms with GPU access, `--headless` prevents a window from spawning while the project is exporting.

```
# `godot` must be a Godot editor binary, not an export
template.
# Also, export templates must be installed for the editor
# (or a valid custom export template must be defined in the
export preset).
godot --headless --export-release "Linux/X11"
/var/builds/project
godot --headless --export-release Android
/var/builds/project.apk
```

The preset name must match the name of an export preset defined in the project's `export_presets.cfg` file. If the preset name contains spaces or special characters (such as "Windows Desktop"), it must be surrounded with quotes.

To export a debug version of the game, use the `--export-debug` switch instead of `--export-release`. Their parameters and usage are the same.

To export only a PCK file, use the `--export-pack` option followed by the preset name and output path, with the file extension, instead of `--export-release` or `--export-debug`. The output path extension determines the package's format, either PCK or ZIP.

Warning

When specifying a relative path as the path for `--export-release`, `--export-debug` or `--export-pack`, the path will be relative to the directory containing the `project.godot` file, **not** relative to the current working directory.

Running a script

It is possible to run a `.gd` script from the command line. This feature is especially useful in large projects, e.g. for batch conversion of assets or custom import/export.

The script must inherit from `SceneTree` or `MainLoop`.

Here is an example `sayhello.gd`, showing how it works:

```
#!/usr/bin/env -S godot -s
extends SceneTree

func _init():
    print("Hello!")
    quit()
```

And how to run it:

```
# Prints "Hello!" to standard output.
godot -s sayhello.gd
```

If no `project.godot` exists at the path, current path is assumed to be the current working directory (unless `--path` is specified).

The first line of `sayhello.gd` above is commonly referred to as a *shebang*. If the Godot binary is in your `PATH` as `godot`, it allows you to run the script as follows in modern Linux distributions, as well as macOS:

```
# Mark script as executable.
chmod +x sayhello.gd
# Prints "Hello!" to standard output.
./sayhello.gd
```

If the above doesn't work in your current version of Linux or macOS, you can always have the shebang run Godot straight from where it is located as follows:

```
#!/usr/bin/godot -s
```

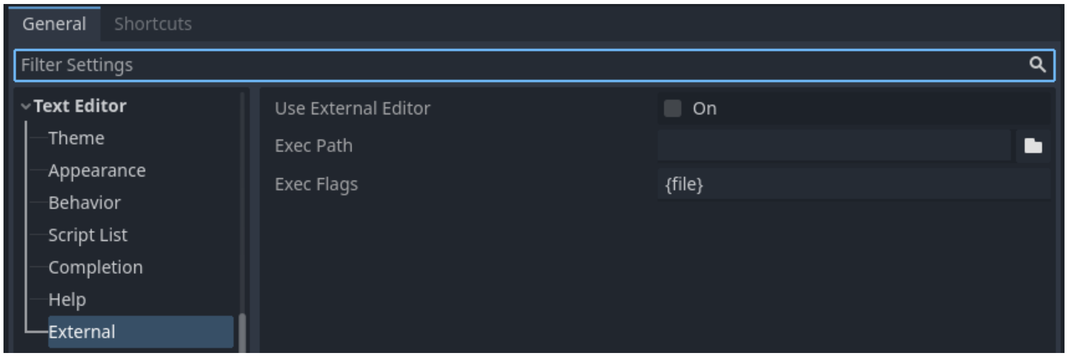
Using an external text editor

This page explains how to code using an external text editor.

Note

To code C# in an external editor, see the C# guide to configure an external editor.

Godot can be used with an external text editor, such as Sublime Text or Visual Studio Code. Browse to the relevant editor settings: **Editor > Editor Settings > Text Editor > External**


General
Shortcuts
Filter Settings
Text Editor
Theme
Appearance
Behavior
Script List
Completion
Help
External

Use External Editor ☐ On

Exec Path

Exec Flags {file}

Text Editor > External section of the Editor Settings

There are two text fields: the executable path and command-line flags. The flags allow you to integrate the editor with Godot, passing it the file path to open and other relevant arguments. Godot will replace the following placeholders in the flags string:

Field in Exec Flags	Is replaced with
`{project}`	The absolute path to the project directory
`{file}`	The absolute path to the file
`{col}`	The column number of the error
`{line}`	The line number of the error

Some example **Exec Flags** for various editors include:

Editor	Exec Flags
Geany/Kate	`{file} --line {line} --column {col}`
Atom	`{file}:{line}`
JetBrains Rider	`{project} --line {line} {file}`
Visual Studio Code	`{project} --goto {file}:{line}:{col}`
Vim (gVim)	`"+call cursor({line}, {col})" {file}`

Editor	**Exec Flags**
Emacs	`emacs +{line}:{col} {file}`
Sublime Text	`{project} {file}:{line}:{column}`

Note

For Visual Studio Code on Windows, you will have to point to the `code.cmd` file.

For Emacs, you can call `emacsclient` instead of `emacs` if you use the server mode.

Using External Editor in Debugger

Using external editor in debugger is determined by a separate option in settings. For details, see Script editor debug tools and options.

Official editor plugins

We have official plugins for the following code editors:

- Visual Studio Code [https://github.com/godotengine/godot-vscode-plugin]
- Emacs [https://github.com/godotengine/emacs-gdscript-mode]

LSP/DAP support

Godot supports the Language Server Protocol [https://microsoft.github.io/language-server-protocol/] (**LSP**) for code completion and the Debug Adapter Protocol [https://microsoft.github.io/debug-adapter-protocol/] (**DAP**) for debugging. You can check the LSP client list [https://microsoft.github.io/language-server-protocol/implementors/tools/] and DAP client list [https://microsoft.github.io/debug-adapter-protocol/implementors/tools/] to find if your editor supports them. If this is the case, you should be able to take advantage of these features without the need of a custom plugin.

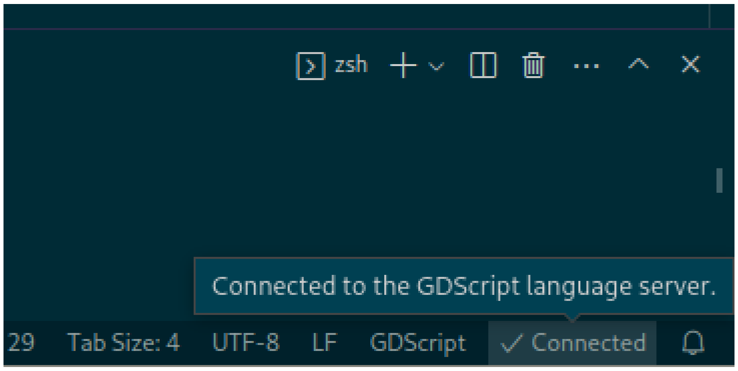
To use these protocols, a Godot instance must be running on your current project. You should then configure your editor to communicate to the running adapter ports in Godot, which by default are `6005` for **LSP**, and `6006` for **DAP**. You can change these ports and other settings in the **Editor Settings**, under the **Network > Language Server** and **Network > Debug Adapter** sections respectively.

Below are some configuration steps for specific editors:

Visual Studio Code

You need to install the official Visual Studio Code plugin [https://github.com/godotengine/godot-vscode-plugin].

For **LSP**, follow these instructions [https://github.com/godotengine/godot-vscode-plugin#gdscript_lsp_server_port] to change the default LSP port. The connection status can be checked on the status bar:



Connected to the GDScript language server.

For **DAP**, specify the `debugServer` property in your `launch.json` file:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "GDScript Godot",
            "type": "godot",
            "request": "launch",
            "project": "${workspaceFolder}",
            "port": 6007,
            "debugServer": 6006,
        }
    ]
}
```

Emacs

Check the official instructions to configure LSP [https://github.com/godotengine/emacs-gdscript-mode#auto-completion-with-the-language-server-protocol-lsp], and DAP [https://github.com/godotengine/emacs-gdscript-mode#using-the-debugger].

JetBrains Rider

Refer to JetBrains Rider documentation [https://www.jetbrains.com/help/rider/Godot.html].

Managing editor features

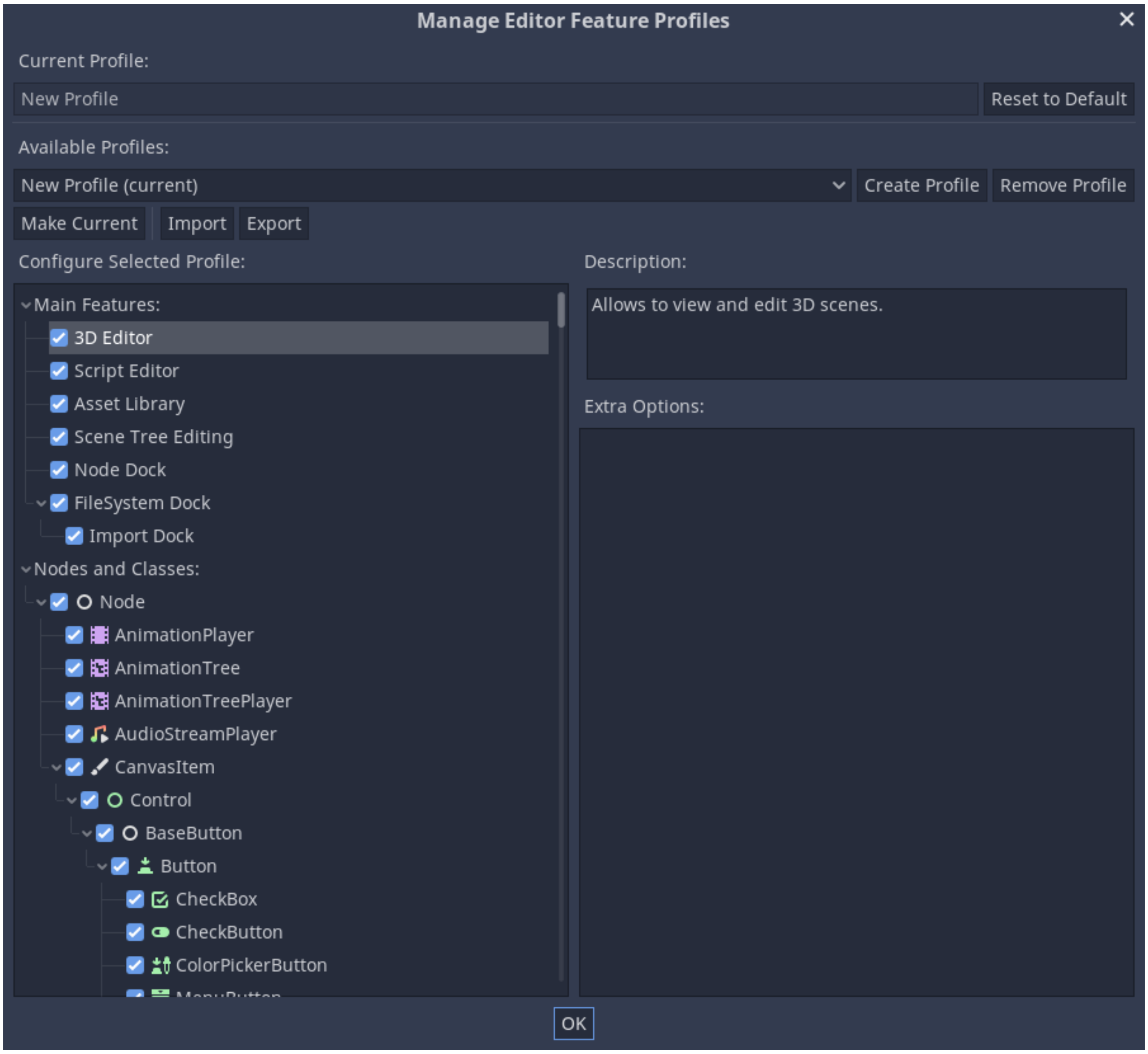
Introduction

In certain situations, it may be desirable to limit what features can be used in the Godot editor. For example, a UI designer on a team who doesn't need to see 3D features, or an educator slowly introducing features to students. Godot has a built in system called "feature profiles" to do this.

With feature profiles, major features and nodes can be hidden from the editor. This only hides parts of the interface and does not actually remove support for these features, so scenes and scripts relying on those features will still work fine. This also means feature profiles are not an optimization technique. For information on how to optimize Godot see Performance.

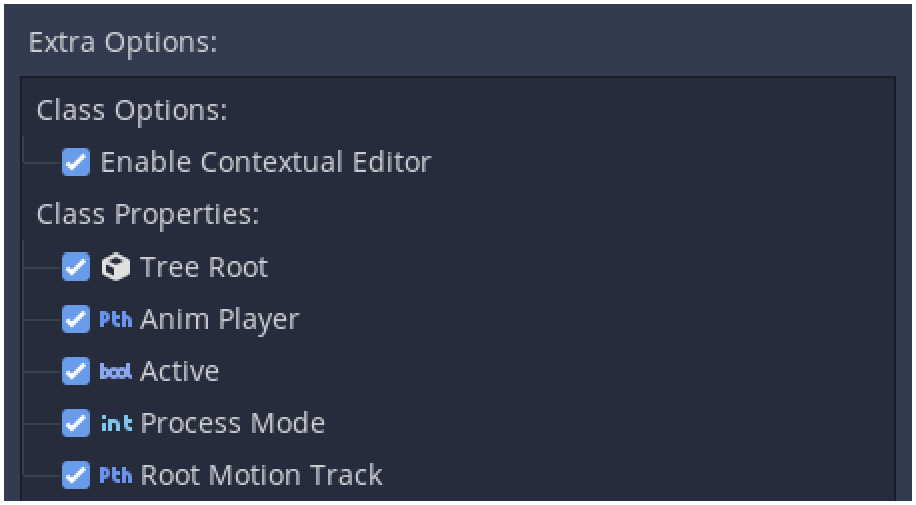
Creating a profile

To manage editor features go to **Editor > Manage Editor Features**. This will open the **Manage Editor Feature Profiles** window. By default there will be no profile. Click on **Create Profile** and give it a name. You will then see a list of all the features in the Godot editor.


Manage Editor Feature Profiles
Current Profile:
New Profile
Reset to Default
Available Profiles:
New Profile (current)
Create Profile
Remove Profile
Make Current
Import
Export
Configure Selected Profile:
Main Features:
☑ 3D Editor
☑ Script Editor
☑ Asset Library
☑ Scene Tree Editing
☑ Node Dock
☑ FileSystem Dock
☑ Import Dock
Nodes and Classes:
☑ Node
☑ AnimationPlayer
☑ AnimationTree
☑ AnimationTreePlayer
☑ AudioStreamPlayer
☑ CanvasItem
☑ Control
☑ BaseButton
☑ Button
☑ CheckBox
☑ CheckButton
☑ ColorPickerButton
Description:
Extra Options:
OK

Description: Allows to view and edit 3D scenes.

The first section allows major editor features to be removed, such as the 3D editor or scripting editor. Below the main features is every class and node in Godot, which can be disabled as well. Click on a node and all of its properties and options will be listed in the **Extra Items** box, these can all be individually disabled.


Extra Options:
Class Options:
☑ Enable Contextual Editor
Class Properties:
☑ Tree Root
☑ Pth Anim Player
☑ bool Active
☑ int Process Mode
☑ Pth Root Motion Track

Sharing a profile

To share profiles between editors click on the **Export** button. Save the custom profile somewhere as a `.profile` file. To use this in another editor open that editor's **Manage Editor Feature Profiles** window and click import, then select the `.profile` file.

This process is potentially cumbersome however if a large amount of computers need custom profiles. As an alternative, you can enable self-contained mode for Godot, which allows putting all editor configuration in the same folder as the editor binary. See Self-contained mode for details.

Migrating to a new version

Godot loosely follows a semantic versioning system, where compatibility is assumed between minor and patch releases, while major releases can break it. As such, it is generally not recommended to move projects between major versions during their development, especially if you've been working on them for a significant amount of time.

Still, new features, usability improvements, or paradigm shifts in engine's internals may incentivize you to upgrade. Below is a list of articles that should assist you when upgrading your project between versions. Each article would try its best to document every important difference and provide you with a migration path.

- Upgrading from Godot 3 to Godot 4
- Upgrading from Godot 4.0 to Godot 4.1
- Upgrading from Godot 4.1 to Godot 4.2
- Upgrading from Godot 4.2 to Godot 4.3

Upgrading from Godot 3 to Godot 4

Should I upgrade to Godot 4?

Before beginning the upgrade process, it's worth thinking about the advantages and disadvantages that upgrading would bring to your project.

Advantages of upgrading

Along with the new features present in 4.0 [https://github.com/godotengine/godot/blob/master/CHANGELOG.md], upgrading gives the following advantages:

- Many bugs are fixed in 4.0, but cannot be resolved in 3.x for various reasons (such as graphics API differences or backwards compatibility).
- 4.x will enjoy a longer support period. Godot 3.x will continue to be supported for some time after 4.0 is released, but it will eventually stop receiving support.

See Documentation changelog for a list of pages documenting new features in Godot 4.0.

Disadvantages of upgrading

If you don't *need* any features present in Godot 4.0, you may want to stay on Godot 3.x for the following reasons:

- Godot 3.x is tried and true, while Godot 4 remains in its early stages. [https://godotengine.org/article/release-management-4-0-and-beyond]
  - Godot 4.0 is expected to contain workflow and performance issues that Godot 3.x doesn't have. These issues will be ironed out over time in future Godot 4.x releases.
- Godot 4 has fewer third-party tutorials available compared to Godot 3.x. If you're new to game engines, you may have a better experience using Godot 3.x as a result.
- Godot 4's baseline hardware requirements (such as memory usage) are slightly higher, both for the editor and exported projects. This was required for the implementation of some core optimizations.
- Since Godot 4 includes more features than Godot 3, Godot 4's binary size for exported projects is larger. While this can be mitigated by optimizing a build for size, a 4.0 build with a given set of enabled modules will remain larger compared to a 3.x build with the same modules. This can be an issue for exporting to the Web,

as binary size directly influences how fast the engine can initialize (regardless of download speed).
- Godot 4 does not and will not have support for GLES2 rendering. (There is still support for GLES3 rendering using the new OpenGL backend, which means that devices without Vulkan support can still run Godot 4.)
  - If you are targeting **very** old hardware such as Intel Sandy Bridge (2nd generation) integrated graphics, this will prevent the project from running on such hardware after upgrading. Software OpenGL implementations [https://github.com/pal1000/mesa-dist-win] can be used to bypass this limitation, but they're too slow for gaming.

Caveats of upgrading

Since Godot 4 is a complete rewrite in many aspects, some features have unfortunately been lost in the process. Some of these features may be restored in future Godot releases:

- Bullet physics was removed in favor of GodotPhysics. This only affects 3D projects that used the default physics engine (which was Bullet) and didn't manually change it to GodotPhysics. There are no plans to re-add Bullet physics in core, but a third-party add-on could be created for it thanks to GDExtension.
- Rendering in 2D is no longer performed in HDR, which means "overbright" modulate values have no visible effect. This is planned to be restored at some point in the future.
- While rendering still happens in HDR in 3D when using the Forward Plus or Forward Mobile backends, Viewports cannot return HDR data anymore. This is planned to be restored at some point in the future.
- Mono was replaced by .NET 6. This means exporting C# projects to Android, iOS and HTML5 is no longer supported for now. Exporting C# projects to desktop platforms is still supported, and as of 4.2 there's experimental support for exporting to mobile platforms. Support for exporting C# projects to more platforms will be restored in future 4.x releases as upstream support improves.

You can find a more complete list of functional regressions by searching for issues labeled "regression" but not "bug" on GitHub [https://github.com/godotengine/godot/issues?q=is%3Aissue+is%3Aopen+label%3Aregression+-label%3Abug].

Preparing before the upgrade (optional)

If you want to be ready to upgrade to Godot 4 in the future, consider using Tweener and the Time singleton in your project. These classes are both available in Godot 3.5 and later.

This way, you won't be relying on the deprecated Tween node and OS time functions, both of which are removed in Godot 4.0.

It's also a good idea to rename external shaders so that their extension is `.gdshader` instead of `.shader`. Godot 3.x supports both extensions, but only `.gdshader` is supported in Godot 4.0.

Running the project upgrade tool

Warning

Make a full backup of your project before upgrading! The project upgrade tool will *not* perform any backups of the project that is being upgraded.

You can backup a project by using version control, or by copying the project folder to another location.

Using the Project Manager

To use the project upgrade tool:

1. Open the Godot 4 Project Manager.
2. Import the Godot 3.x project using the **Import** button, or use the **Scan** button to find the project within a folder.
3. Double-click the imported project (or select the project then choose **Edit**).
4. You will see a dialog appearing with two options: **Convert project.godot Only** and **Convert Full Project**. After ensuring your project is backed up (see the above warning), choose **Convert Full Project**. **Convert project.godot Only** is intended to be used for advanced use cases *only*, in case the conversion tool fails.
5. Wait until the project conversion process finishes. This can take up to a few minutes for large projects with lots of scenes.
6. When the Project Manager interface becomes available again, double-click the project (or select the project then choose **Edit**) to open it in the editor.

If you hit conversion issues due to some project files being too large or long, you can use the command line to upgrade the project (see below). This will allow you to override the converter's size limits.

Using the command line

To use the project upgrade tool from the command line, it's recommended to validate the project conversion by running the Godot editor binary with the following arguments:

```
# [<max_file_kb>] [<max_line_size>] are optional arguments.
# Remove them if you aren't changing their values.
path/to/godot.binary --path /path/to/project/folder --validate-conversion-3to4
[<max_file_kb>] [<max_line_size>]
```

If the list of planned upgrades looks good to you, run the following command on the Godot editor binary to upgrade project files:

```
# [<max_file_kb>] [<max_line_size>] are optional arguments.
# Remove them if you aren't changing their values.
path/to/godot.binary --path /path/to/project/folder --convert-3to4
[<max_file_kb>] [<max_line_size>]
```

`[<max_file_kb>]` and `[<max_line_size>]` are *optional* arguments to specify the maximum size of files to be converted (in kilobytes and lines). The default limits are 4 MB and 100,000 lines respectively. If a file hits either of those limits, it will not be upgraded by the project converter. This is useful to prevent large resources from slowing down the upgrade to a crawl.

If you still want large files to be converted by the project upgrade tool, increase the size limits when running the project upgrade tool. For example, running the Godot editor binary with those arguments increases both limits by a 10× factor:

```
path/to/godot.binary --path /path/to/project/folder --convert-3to4 40000 1000000
```

Note

Only Godot 3.0 and later projects can be upgraded using the project conversion tool found in the Godot 4 editor.

It's recommended to ensure that your project is up-to-date with the latest 3.x stable release before running the project upgrade tool.

Fixing the project after running the project upgrade tool

After upgrading the project, you may notice that certain things don't look as they should. Scripts will likely contain various errors as well (possibly hundreds in large projects). This is because the project upgrade tool cannot cater to all situations. Therefore, a large part of the upgrade process remains manual.

Automatically renamed nodes and resources

The list below refers to nodes which were simply renamed for consistency or clarity in Godot 4.0. The project upgrade tool renames them automatically in your scripts.

One noteworthy set of renames is 3D nodes, which all got a `3D` suffix added for consistency with their 2D counterparts. For example, `Area` is now `Area3D`.

For ease of searching, this table lists all nodes and resources that were renamed and are automatically converted, excluding the ones which only involved adding a `3D` suffix to the old name:

Old name (Godot 3.x)	New name (Godot 4)
AnimatedSprite	AnimatedSprite2D
ARVRCamera	XRCamera3D
ARVRController	XRController3D
ARVRAnchor	XRAnchor3D
ARVRInterface	XRInterface
ARVROrigin	XROrigin3D
ARVRPositionalTracker	XRPositionalTracker
ARVRServer	XRServer
CubeMesh	BoxMesh
EditorSpatialGizmo	EditorNode3DGizmo
EditorSpatialGizmoPlugin	EditorNode3DGizmoPlugin
GIProbe	VoxelGI
GIProbeData	VoxelGIData
GradientTexture	GradientTexture1D

Old name (Godot 3.x)	New name (Godot 4)
KinematicBody	CharacterBody3D
KinematicBody2D	CharacterBody2D
Light2D	PointLight2D
LineShape2D	WorldBoundaryShape2D
Listener	AudioListener3D
NavigationMeshInstance	NavigationRegion3D
NavigationPolygonInstance	NavigationRegion2D
Navigation2DServer	NavigationServer2D
PanoramaSky	Sky
Particles	GPUParticles3D
Particles2D	GPUParticles2D
ParticlesMaterial	ParticleProcessMaterial
Physics2DDirectBodyState	PhysicsDirectBodyState2D
Physics2DDirectSpaceState	PhysicsDirectSpaceState2D
Physics2DServer	PhysicsServer2D

Old name (Godot 3.x)	New name (Godot 4)
Physics2DShapeQueryParameters	PhysicsShapeQueryParameters2D
Physics2DTestMotionResult	PhysicsTestMotionResult2D
PlaneShape	WorldBoundaryShape3D
Position2D	Marker2D
Position3D	Marker3D
ProceduralSky	Sky
RayShape	SeparationRayShape3D
RayShape2D	SeparationRayShape2D
ShortCut	Shortcut
Spatial	Node3D
SpatialGizmo	Node3DGizmo
SpatialMaterial	StandardMaterial3D
Sprite	Sprite2D
StreamTexture	CompressedTexture2D
TextureProgress	TextureProgressBar

Old name (Godot 3.x)	New name (Godot 4)
VideoPlayer	VideoStreamPlayer
ViewportContainer	SubViewportContainer
Viewport	SubViewport
VisibilityEnabler	VisibleOnScreenEnabler3D
VisibilityNotifier	VisibleOnScreenNotifier3D
VisibilityNotifier2D	VisibleOnScreenNotifier2D
VisibilityNotifier3D	VisibleOnScreenNotifier3D
VisualServer	RenderingServer
VisualShaderNodeScalarConstant	VisualShaderNodeFloatConstant
VisualShaderNodeScalarFunc	VisualShaderNodeFloatFunc
VisualShaderNodeScalarOp	VisualShaderNodeFloatOp
VisualShaderNodeScalarClamp	VisualShaderNodeClamp
VisualShaderNodeVectorClamp	VisualShaderNodeClamp
VisualShaderNodeScalarInterp	VisualShaderNodeMix
VisualShaderNodeVectorInterp	VisualShaderNodeMix

Old name (Godot 3.x)	New name (Godot 4)
VisualShaderNodeVectorScalarMix	VisualShaderNodeMix
VisualShaderNodeScalarSmoothStep	VisualShaderNodeSmoothStep
VisualShaderNodeVectorSmoothStep	VisualShaderNodeSmoothStep
VisualShaderNodeVectorScalarSmoothStep	VisualShaderNodeSmoothStep
VisualShaderNodeVectorScalarStep	VisualShaderNodeStep
VisualShaderNodeScalarSwitch	VisualShaderNodeSwitch
VisualShaderNodeScalarTransformMult	VisualShaderNodeTransformOp
VisualShaderNodeScalarDerivativeFunc	VisualShaderNodeDerivativeFunc
VisualShaderNodeVectorDerivativeFunc	VisualShaderNodeDerivativeFunc
VisualShaderNodeBooleanUniform	VisualShaderNodeBooleanParameter
VisualShaderNodeColorUniform	VisualShaderNodeColorParameter
VisualShaderNodeScalarUniform	VisualShaderNodeFloatParameter
VisualShaderNodeCubeMapUniform	VisualShaderNodeCubeMapParameter
VisualShaderNodeTextureUniform	VisualShaderNodeTexture2DParameter
VisualShaderNodeTextureUniformTriplanar	VisualShaderNodeTextureParameterTriplanar

Old name (Godot 3.x)	New name (Godot 4)
VisualShaderNodeTransformUniform	VisualShaderNodeTransformParameter
VisualShaderNodeVec3Uniform	VisualShaderNodeVec3Parameter
VisualShaderNodeUniform	VisualShaderNodeParameter
VisualShaderNodeUniformRef	VisualShaderNodeParameterRef

Manually renaming methods, properties, signals and constants

Due to how the project upgrade tool works, not all API renames can be performed automatically. The list below contains all renames that must be performed manually using the script editor.

If you cannot find a node or resource in the list below, refer to the above table to find its new name.

Tip

You can use the **Replace in Files** dialog to speed up replacement by pressing `Ctrl + Shift + R` while the script editor is open. However, be careful as the Replace in Files dialog doesn't offer any way to undo a replacement. Use version control to commit your upgrade work regularly. Command line tools such as sd [https://github.com/chmln/sd] can also be used if you need something more flexible than the editor's Replace in Files dialog.

If using C#, remember to search for outdated API usage with PascalCase notation in the project (and perform the replacement with PascalCase notation).

Methods

- File and Directory classes were replaced by FileAccess and DirAccess, which have an entirely different API. Several methods are now static, which means you can call them directly on FileAccess or DirAccess without having to create an instance of that class.

- Screen and window-related methods from the OS singleton (such as `OS.get_screen_size()`) were moved to the DisplayServer singleton. Method naming was also changed to use the `DisplayServer.<object>_<get/set>_property()` form instead. For example, `OS.get_screen_size()` becomes `DisplayServer.screen_get_size()`.
- Time and date methods from the OS singleton were moved to the Time singleton. (The Time singleton is also available in Godot 3.5 and later.)
- You may have to replace some `instance()` calls with `instantiate()`. The converter *should* handle this automatically, but this relies on custom code that may not work in 100% of situations.
- AcceptDialog's `set_autowrap()` is now `set_autowrap_mode()`.
- AnimationNode's `process()` is now `_process()` (note the leading underscore, which denotes a virtual method).
- AStar2D and AStar3D's `get_points()` is now `get_points_id()`.
- BaseButton's `set_event()` is now `set_shortcut()`.
- Camera2D's `get_v_offset()` is now `get_drag_vertical_offset()`.
- Camera2D's `set_v_offset()` is now `set_drag_vertical_offset()`.
- CanvasItem's `update()` is now `queue_redraw()`.
- Control's `set_tooltip()` is now `set_tooltip_text()`.
- EditorNode3DGizmoPlugin's `create_gizmo()` is now `_create_gizmo()` (note the leading underscore, which denotes a virtual method).
- ENetMultiplayerPeer's `get_peer_port()` is now `get_peer()`.
- FileDialog's `get_mode()` is now `get_file_mode()`.
- FileDialog's `set_mode()` is now `set_file_mode()`.
- GraphNode's `get_offset()` is now `get_position_offset()`.
- GridMap's `world_to_map()` is now `local_to_map()`.
- GridMap's `map_to_world()` is now `map_to_local()`.
- Image's `get_rect()` is now `get_region()`.
- ItemList's `get_v_scroll()` is now `get_v_scroll_bar()`.
- MultiPlayerAPI's `get_network_connected_peers()` is now `get_peers()`.
- MultiPlayerAPI's `get_network_peer()` is now `get_peer()`.
- MultiPlayerAPI's `get_network_unique_id()` is now `get_unique_id()`.
- MultiPlayerAPI's `has_network_peer()` is now `has_multiplayer_peer()`.
- PacketPeerUDP's `is_listening()` is now `is_bound()`.
- PacketPeerUDP's `listen()` is now `bind()`.
- ParticleProcessMaterial's `set_flag()` is now `set_particle_flag()`.
- ResourceFormatLoader's `get_dependencies()` is now `_get_dependencies()` (note the leading underscore, which denotes a virtual method).
- SceneTree's `change_scene()` is now `change_scene_to_file()`.
- Shortcut's `is_valid()` is now `has_valid_event()`.
- TileMap's `world_to_map()` is now `local_to_map()`.
- TileMap's `map_to_world()` is now `map_to_local()`.
- Transform2D's `xform()` is `mat * vec` and `xform_inv()` is `vec * mat`.

Properties

Note

If a property is listed here, its associated getter and setter methods must also be renamed manually if used in the project. For example, PathFollow2D and PathFollow3D's `set_offset()` and `get_offset()` must be renamed to `set_progress()` and `get_progress()` respectively.

- Control's `margin` is now `offset`.
- Label's `percent_visible` is now `visible_ratio`.
- MultiPlayerAPI's `refuse_new_network_connections` is now `refuse_new_connections`.
- PathFollow2D and PathFollow3D's `offset` is now `progress`.
- TextureProgressBar's `percent_visible` is now `show_percentage`.
- The `extents` property on CSG nodes and VoxelGI will have to be replaced with `size`, with the set value halved (as they're no longer half-extents). This also affects its setter/getter methods `set_extents()` and `get_extents()`.
- The `Engine.editor_hint` property was removed in favor of the `Engine.is_editor_hint()` *method*. This is because it's read-only, and properties in Godot are not used for read-only values.

Enums

- CPUParticles2D's `FLAG_MAX` is now `PARTICLE_FLAG_MAX`.

Signals

- FileSystemDock's `instantiate` is now `instance`.
- CanvasItem's `hide` is now `hidden`. This rename does **not** apply to the `hide()` method, only the signal.
- Tween's `tween_all_completed` is now `loop_finished`.
- EditorSettings' `changed` is now `settings_changed`.

Constants

- Color names are now uppercase and use underscores between words. For example, `Color.palegreen` is now `Color.PALE_GREEN`.
- MainLoop's `NOTIFICATION_` constants were duplicated to `Node` which means you can remove the `MainLoop.` prefix when referencing them.
- MainLoop's `NOTIFICATION_WM_QUIT_REQUEST` is now `NOTIFICATION_WM_CLOSE_REQUEST`.

Checking project settings

Several project settings were renamed, and some of them had their enums changed in incompatible ways (such as shadow filter quality). This means you may need to set some project settings' values again. Make sure the **Advanced** toggle is enabled in the project settings dialog so you can see all project settings.

Checking Environment settings

Graphics quality settings were moved from Environment properties to project settings. This was done to make run-time quality adjustments easier, without having to access the currently active Environment resource then modify its properties.

As a result, you will have to configure Environment quality settings in the project settings as old Environment quality settings aren't converted automatically to project settings.

If you have a graphics settings menu that changed environment properties in Godot 3.x, you will have to change its code to call RenderingServer methods that affect environment effects' quality. Only the "base" toggle of each environment effect and its visual knobs remain within the Environment resource.

Updating shaders

There have been some changes to shaders that aren't covered by the upgrade tool.

The `.shader` file extension is no longer supported, which means you must rename `.shader` files to `.gdshader` and update references accordingly in scene/resource files using an external text editor.

Some notable renames you will need to perform in shaders are:

- Texture filter and repeat modes are now set on individual uniforms, rather than the texture files themselves.
- `hint_albedo` is now `source_color`.
- `hint_color` is now `source_color`.
- Built in matrix variables were renamed.
- Particles shaders no longer use the `vertex()` processor function. Instead they use `start()` and `process()`.

See Shading language for more information.

Updating scripts to take backwards-incompatible changes into account

Some changes performed between Godot 3.x and 4 are not renames, but they still break backwards compatibility due to different default behavior.

The most notable examples of this are:

- Lifecycle functions such as `_ready()` and `_process()` no longer implicitly call parent classes' functions that have the same name. Instead, you must use `super()` at the top of a lifecycle function in the child class so that the parent class function is called first.
- Both String and StringName are now exposed to GDScript. This allows for greater optimization, as StringName is specifically designed to be used for "constant" strings that are created once and reused many times. These types are not strictly equivalent to each other, which means `is_same("example", &"example")` returns `false`. Although in most cases they are interchangeable (`"example" == &"example"` returns `true`), sometimes you may have to replace `"example"` with `&"example"`.
- GDScript setter and getter syntax was changed, but it's only partially converted by the conversion tool. In most cases, manual changes are required to make setters and getters working again.
- GDScript signal connection syntax was changed. The conversion tool will use the string-based syntax which is still present in Godot 4, but it's recommended to switch to the Signal-based syntax described on the linked page. This way, strings are no longer involved, which avoids issues with signal name errors that can only be discovered at run-time.
- Built-in scripts that are tool scripts do not get the `tool` keyword converted to the `@tool` annotation.
- The Tween node was removed in favor of Tweeners, which are also available in Godot 3.5 and later. See the original pull request [https://github.com/godotengine/godot/pull/41794] for details.
- `randomize()` is now automatically called on project load, so deterministic randomness with the global RandomNumberGenerate instance requires manually setting a seed in a script's `_ready()` function.
- `call_group()`, `set_group()` and `notify_group()` are now immediate by default. If calling an expensive function, this may result in stuttering when used on a group containing a large number of nodes. To use deferred calls like before, replace `call_group(...)` with `call_group_flags(SceneTree.GROUP_CALL_DEFERRED, ...)` (and do the same with `set_group()` and `notify_group()` respectively).
- Instead of `rotation_degrees`, the `rotation` property is exposed to the editor, which is automatically displayed as degrees in the Inspector dock. This may break animations, as the conversion is not handled automatically by the conversion tool.
- AABB's `has_no_surface()` was inverted and renamed to `has_surface()`.
- AABB and Rect2's `has_no_area()` was inverted and renamed to `has_area()`.
- AnimatedTexture's `fps` property was replaced by `speed_scale`, which works the same as AnimationPlayer's `playback_speed` property.
- AnimatedSprite2D and AnimatedSprite3D now allow negative `speed_scale` values. This may break animations if you relied on `speed_scale` being internally clamped to `0.0`.
- AnimatedSprite2D and AnimatedSprite3D's `playing` property was removed. Use `play()`/`stop()` method instead OR configure `autoplay` animation via the

SpriteFrames bottom panel (but not both at once).

- Array's `slice()` second parameter (`end`) is now *exclusive*, instead of being inclusive. For example, this means that `[1, 2, 3].slice(0, 1)` now returns `[1]` instead of `[1, 2]`.
- BaseButton's signals are now `button_up` and `button_down`. The `pressed` property is now `button_pressed`.
- Camera2D's `rotating` property was replaced by `ignore_rotation`, which has inverted behavior.
- Camera2D's `zoom` property was inverted: higher values are now more zoomed in, instead of less.
- Node's `remove_and_skip()` method was removed. If you need to reimplement it in a script, you can use the old C++ implementation [https://github.com/godotengine/godot/blob/7936b3cc4c657e4b273b376068f095e1e0e4d82a/scene/main/node.cpp#L1910-L1945] as a reference.
- `OS.get_system_time_secs()` should be converted to `Time.get_time_dict_from_system()["second"]`.
- ResourceSaver's `save()` method now has its arguments swapped around (`resource: Resource, path: String`). This also applies to ResourceFormatSaver's `_save()` method.
- A StreamPeerTCP must have `poll()` called on it to update its state, instead of relying on `get_status()` automatically polling: GH-59582 [https://github.com/godotengine/godot/pull/59582]
- String's `right()` method has changed behavior [https://github.com/godotengine/godot/pull/36180]: it now returns a number of characters from the right of the string, rather than the right side of the string from a given position. If you need the old behavior, you can use `substr()` instead.
- `is_connected_to_host()` was removed from StreamPeerTCP and PacketPeerUDP as per GH-59582 [https://github.com/godotengine/godot/pull/59582]. `get_status()` can be used in StreamPeerTCP instead. `is_socket_connected()` can be used in PacketPeerUDP instead.
- In `_get_property_list()`, the `or_lesser` property hint string is now `or_less`.
- In `_get_property_list()`, the `noslider` property hint string is now `no_slider`.
- VisualShaderNodeVec4Parameter now takes a Vector4 as parameter instead of a Quaternion.

Removed or replaced nodes/resources

This lists all nodes that were replaced by another node requiring different configuration. The setup must be done from scratch again, as the project converter doesn't support updating existing setups:

Removed node	Closest approximation	Comment
AnimationTreePlayer	AnimationTree	AnimationTreePlayer was deprecated since Godot 3.1.
BakedLightmap	LightmapGI	See Using Lightmap global illumination.
BakedLightmapData	LightmapGIData	
BitmapFont	FontFile	See Using Fonts.
DynamicFont	FontFile	
DynamicFontData	FontFile	
Navigation2D	Node2D	Replaced by other 2D Navigation nodes.
Navigation3D	Node3D	Replaced by other 3D Navigation nodes.
OpenSimplexNoise	FastNoiseLite	Has different parameters and more noise types such as cellular. No support for 4D noise as it's absent from the FastNoiseLite library.
ToolButton	Button	ToolButton was Button with the **Flat** property enabled by default.
YSort	Node2D or Control	CanvasItem has a new **Y Sort Enabled** property in 4.0.
ProximityGroup	Node3D	VisibleOnScreenNotifier3D can act as a replacement.

Removed node	Closest approximation	Comment
Portal	Node3D	Portal and room occlusion culling was replaced by raster occlusion culling (OccluderInstance3D node), which requires a different setup process.
Room	Node3D	
RoomManager	Node3D	
RoomGroup	Node3D	
Occluder	Node3D	Geometry occlusion culling was replaced by raster occlusion culling (OccluderInstance3D node), which requires a different setup process.
OccluderShapeSphere	Resource	

If loading an old project, the node will be replaced with its *Closest approximation* automatically (even if not using the project upgrade tool).

Threading changes

Threading APIs have changed in 4.0. For example, the following code snippet in Godot 3.x must be modified to work in 4.0:

```
# 3.x
var start_success = new_thread.start(self, "__threaded_background_loader",
    [resource_path, thread_num]
)

# 4.0
var start_success =
new_thread.start(__threaded_background_loader.bind(resource_path, thread_num))
```

`Thread.is_active()` is no longer used and should be converted to `Thread.is_alive()`.

See also

See the changelog [https://github.com/godotengine/godot/blob/master/CHANGELOG.md] for a full list of changes between Godot 3.x and 4.

ArrayMesh resource compatibility breakage

If you've saved an ArrayMesh resource to a `.res` or `.tres` file, the format used in 4.0 is not compatible with the one used in 3.x. You will need to go through the process of importing the source mesh file and saving it as an ArrayMesh resource again.

List of automatically renamed methods, properties, signals and constants

The editor/renames_map_3_to_4.cpp [https://github.com/godotengine/godot/blob/master/editor/renames_map_3_to_4.cpp] source file lists all automatic renames performed by the project upgrade tool. Lines that are commented out refer to API renames that cannot be performed automatically.

Porting editor settings

Godot 3.x and 4.0 use different editor settings files. This means their settings can be changed independently from each other.

If you wish to port over your Godot 3.x settings to Godot 4, open the editor settings folder and copy `editor_settings-3.tres` to `editor_settings-4.tres` while the Godot 4 editor is closed.

Note

Many settings' names and categories have changed since Godot 3.x. Editor settings whose name or category has changed won't carry over to Godot 4.0; you will have to set their values again.

Upgrading from Godot 4.0 to Godot 4.1

For most games and apps made with 4.0, it should be relatively safe to migrate to 4.1. This page intends to cover everything you need to pay attention to when migrating your project.

Breaking changes

If you are migrating from 4.0 to 4.1, the breaking changes listed here might affect you. Changes are grouped by areas/systems.

Warning

The GDExtension API completely breaks compatibility in 4.1, so it's not included in the table below. See the Updating your GDExtension for 4.1 section for more information.

This article indicates whether each breaking change affects GDScript and whether the C# breaking change is *binary compatible* or *source compatible*:

- **Binary compatible** - Existing binaries will load and execute successfully without recompilation, and the run-time behavior won't change.
- **Source compatible** - Source code will compile successfully without changes when upgrading Godot.

Core

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Basis				
Method `looking_at` adds a new `use_model_front` optional parameter	✓	✓	✓	GH-76082
Object				
Method `get_meta_list` changes return type from `PackedStringArray` to `Array[StringName]`	✓	✗	✗	GH-76418
Transform3D				

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Method `looking_at` adds a new `use_model_front` optional parameter	✓	✓	✓	[GH-76082]()
UndoRedo				
Method `create_action` adds a new `backward_undo_ops` optional parameter	✓	✓	✓	[GH-76688]()
WorkerThreadPool				
Method `wait_for_task_completion` changes return type from `void` to `Error`	✓	✗	✓	[GH-77143]()

Animation

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
AnimationNode				
Method `_process` adds a new `test_only` parameter	✗	✗	✗	[GH-75759]()
Method `blend_input` adds a new `test_only` optional parameter	✓	✓	✓	[GH-75759]()
Method `blend_node` adds a new `test_only` optional parameter	✓	✓	✓	[GH-75759]()
AnimationNodeStateMachinePlayback				
Method `get_travel_path` changes return type from `PackedStringArray` to `Array[StringName]`	✓	✗	✗	[GH-76418]()

2D nodes

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
PathFollow2D				
Property `lookahead` removed	❌	❌	❌	GH-72842

3D nodes

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Geometry3D				
Method `segment_intersects_convex` changes `planes` parameter type from untyped `Array` to `Array[Plane]`	✓	✓	❌	GH-76418
MeshInstance3D				
Method `create_multiple_convex_collisions` adds a new `settings` optional parameter	✓	✓	✓	GH-72152
Node3D				
Method `look_at` adds a new `use_model_front` optional parameter	✓	✓	✓	GH-76082
Method `look_at_from_position` adds a new `use_model_front` optional parameter	✓	✓	✓	GH-76082

GUI nodes

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
CodeEdit				
Method `add_code_completion_option` adds a new `location` optional parameter	✓	✓	✓	GH-75746
RichTextLabel				
Method `push_list` adds a new `bullet` optional parameter	✓	✓	✓	GH-75017
Method `push_paragraph` adds a new `justification_flags` optional parameter	✓	✓	✓	GH-75250
Method `push_paragraph` adds a new `tab_stops` optional parameter	✓	✓	✓	GH-76401
Tree				
Method `edit_selected` adds a new `force_edit` optional parameter	✓	✓	✓	GH-76794

Physics

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Area2D				
Property `priority` changes type from `float` to `int`	✗	✗	✗	GH-72749
Area3D				
Property `priority` changes type from `float` to `int`	✗	✗	✗	GH-72749

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
PhysicsDirectSpaceState2D				
Method `collide_shape` changes return type from `Array[PackedVector2Array]` to `Array[Vector2]`	❌	❌	❌	[GH-75260]()
PhysicsDirectSpaceState3D				
Method `collide_shape` changes return type from `Array[PackedVector3Array]` to `Array[Vector3]`	❌	❌	❌	[GH-75260]()

Rendering

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
RDShaderFile				
Method `get_version_list` changes return type from `PackedStringArray` to `Array[StringName]`	✔️	❌	❌	[GH-76418]()
RenderingDevice				
Method `draw_list_begin` changes `storage_textures` parameter type from untyped `Array` to `Array[RID]`	✔️	✔️	❌	[GH-76418]()
RenderingServer				
Method `global_shader_parameter_get_list` changes return type from `PackedStringArray` to `Array[StringName]`	✔️	❌	❌	[GH-76418]()
SurfaceTool				

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Method `add_triangle_fan` changes `tangents` parameter type from untyped `Array` to `Array[Plane]`	✓	✓	✗	GH-76418

Navigation

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
NavigationAgent2D				
Method `set_velocity` replaced with `velocity` property	✓	✗	✗	GH-69988
Property `time_horizon` split into `time_horizon_agents` and `time_horizon_obstacles`	✗	✗	✗	GH-69988
NavigationAgent3D				
Property `agent_height_offset` renamed to `path_height_offset`	✗	✗	✗	GH-69988
Property `ignore_y` removed	✗	✗	✗	GH-69988
Method `set_velocity` replaced with `velocity` property	✓	✗	✗	GH-69988
Property `time_horizon` split into `time_horizon_agents` and `time_horizon_obstacles`	✗	✗	✗	GH-69988
NavigationObstacle2D				
Property `estimate_radius` removed	✗	✗	✗	GH-69988

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Method `get_rid` renamed to `get_agent_rid`	❌	❌	❌	GH-69988
NavigationObstacle3D				
Property `estimate_radius` removed	❌	❌	❌	GH-69988
Method `get_rid` renamed to `get_agent_rid`	❌	❌	❌	GH-69988
NavigationServer2D				
Method `agent_set_callback` renamed to `agent_set_avoidance_callback`	❌	❌	❌	GH-69988
Method `agent_set_target_velocity` removed	❌	❌	❌	GH-69988
Method `agent_set_time_horizon` split into `agent_set_time_horizon_agents` and `agent_set_time_horizon_obstacles`	❌	❌	❌	GH-69988
NavigationServer3D				
Method `agent_set_callback` renamed to `agent_set_avoidance_callback`	❌	❌	❌	GH-69988
Method `agent_set_target_velocity` removed	❌	❌	❌	GH-69988
Method `agent_set_time_horizon` split into `agent_set_time_horizon_agents` and `agent_set_time_horizon_obstacles`	❌	❌	❌	GH-69988

Networking

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
WebRTCPeerConnectionExtension				
Method `_create_data_channel` changes return type from `Object` to `WebRTCDataChannel`	✓	✗	✓	GH-78237

Editor plugins

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
AnimationTrackEditPlugin				
Type `AnimationTrackEditPlugin` removed	✗	✗	✗	GH-76413
EditorInterface				
Type `EditorInterface` changes inheritance from `Node` to `Object`	✓	✗	✗	GH-76176
Method `set_movie_maker_enabled` replaced with `movie_maker_enabled` property	✓	✗	✗	GH-76176
Method `is_movie_maker_enabled` replaced with `movie_maker_enabled` property	✓	✗	✗	GH-76176
EditorResourcePreviewGenerator				
Method `_generate` adds a new `metadata` parameter	✗	✗	✗	GH-64628
Method `_generate_from_path` adds a new `metadata` parameter	✗	✗	✗	GH-64628
EditorUndoRedoManager				

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Method `create_action` adds a new `backward_undo_ops` optional parameter	✓	✓	✓	GH-76688

Behavior changes

In 4.1 some behavior changes have been introduced, which might require you to adjust your project.

Change	Introduced
SubViewportContainer	
When input events should reach SubViewports and their children, `SubViewportContainer.mouse_filter` now needs to be `MOUSE_FILTER_STOP` or `MOUSE_FILTER_PASS`. See GH-79271 for details.	GH-57894
Multiple layered `SubViewportContainer` nodes, that should all receive mouse input events, now need to be replaced by `Area2D` nodes. See GH-79128 for details.	GH-57894
Viewport	
`Viewport` nodes, that have Physics Picking enabled, now automatically set InputEvents as handled. See GH-79897 for workarounds.	GH-77595

Updating your GDExtension for 4.1

GDExtension is still in beta. Until it's marked as stable, compatibility may break when upgrading to a new minor version of Godot.

In order to fix a serious bug, in Godot 4.1 we had to break binary compatibility in a big way and source compatibility in a small way.

This means that GDExtensions made for Godot 4.0 will need to be recompiled for Godot 4.1 (using the `4.1` branch of godot-cpp), with a small change to their source code.

In Godot 4.0, your "entry_symbol" function looks something like this:

```
GDExtensionBool GDE_EXPORT example_library_init(const GDExtensionInterface *p_interface,
const GDExtensionClassLibraryPtr p_library, GDExtensionInitialization *r_initialization) {
    godot::GDExtensionBinding::InitObject init_obj(p_interface, p_library, r_initialization);
    init_obj.register_initializer(initialize_example_module);
    init_obj.register_terminator(uninitialize_example_module);
    init_obj.set_minimum_library_initialization_level(MODULE_INITIALIZATION_LEVEL_SCENE);

    return init_obj.init();
}
```

However, for Godot 4.1, it should look like:

```
GDExtensionBool GDE_EXPORT example_library_init(GDExtensionInterfaceGetProcAddress
p_get_proc_address, const GDExtensionClassLibraryPtr p_library, GDExtensionInitialization
*r_initialization) {
    godot::GDExtensionBinding::InitObject init_obj(p_get_proc_address, p_library,
r_initialization);

    init_obj.register_initializer(initialize_example_module);
    init_obj.register_terminator(uninitialize_example_module);
    init_obj.set_minimum_library_initialization_level(MODULE_INITIALIZATION_LEVEL_SCENE);

    return init_obj.init();
}
```

There are two small changes:

1. The first argument changes from `const GDExtensionInterface *p_interface` to `GDExtensionInterfaceGetProcAddress p_get_proc_address`
2. The constructor for the *init_obj* variable now receives `p_get_proc_address` as its first parameter

You also need to add an extra `compatibility_minimum` line to your `.gdextension` file, so that it looks something like:

```
[configuration]

entry_symbol = "example_library_init"
compatibility_minimum = 4.1
```

This lets Godot know that your GDExtension has been updated and is safe to load in Godot 4.1.

Upgrading from Godot 4.1 to Godot 4.2

For most games and apps made with 4.1 it should be relatively safe to migrate to 4.2. This page intends to cover everything you need to pay attention to when migrating your project.

Breaking changes

If you are migrating from 4.1 to 4.2, the breaking changes listed here might affect you. Changes are grouped by areas/systems.

Warning

The Mesh resource format has changed in 4.2 to allow for vertex and attribute compression [https://github.com/godotengine/godot/pull/81138]. This allows for improved rendering performance, especially on platforms constrained by memory bandwidth such as mobile.

It is still possible to load the Godot 4.0-4.1 Mesh formats, but it is **not** possible to load the Godot 4.2 Mesh format in prior Godot versions. When opening a Godot project made with a version prior to 4.2, you may be presented with an upgrade dialog that offers two options:

- **Restart & Upgrade:** Upgrades the mesh format for all meshes in the project and saves the result to disk. Once chosen, this option prevents downgrading the project to a Godot version prior to 4.2. Set up a version control system and push your changes *before* choosing this option!
- **Upgrade Only:** Upgrades the mesh format in-memory without writing it to disk. This allows downgrading the project to a Godot version older than 4.2 if you need to do so in the future. The downside is that loading the project will be slower every time as the mesh format needs to be upgraded every time the project is loaded. These increased loading times will also affect the exported project. The number and complexity of Mesh resources determines how much loading times are affected.

If this dialog doesn't appear, use **Project > Tools > Upgrade Mesh Surfaces...** at the top of the editor.

This article indicates whether each breaking change affects GDScript and whether the C# breaking change is *binary compatible* or *source compatible*:

- **Binary compatible** - Existing binaries will load and execute successfully without recompilation, and the run-time behavior won't change.
- **Source compatible** - Source code will compile successfully without changes when upgrading Godot.

Core

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Node				
Constant `NOTIFICATION_NODE_RECACHE_REQUESTED` removed	✗	✓	✗	GH-84419

Animation

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
AnimationPlayer				
Method `_post_process_key_value` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `add_animation_library` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `advance` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Signal `animation_finished` moved to base class `AnimationMixer`	✓	✗	✗	GH-80813
Signal `animation_started` moved to base class `AnimationMixer`	✓	✗	✗	GH-80813
Signal `animation_libraries_updated` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Signal `animation_list_changed` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813

Change	**GDScript Compatible**	**C# Binary Compatible**	**C# Source Compatible**	**Introduced**
Property `audio_max_polyphony` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Signal `caches_cleared` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `clear_caches` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `find_animation` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `find_animation_library` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `get_animation` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `get_animation_library` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `get_animation_library_list` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `get_animation_list` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `has_animation` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `has_animation_library` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Property `method_call_mode` renamed to `callback_mode_method` and moved to base class `AnimationMixer`	✓	✓	✓	GH-80813

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Property `playback_active` renamed to `active` and moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Property `playback_process_mode` renamed to `callback_mode_process` and moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `remove_animation_library` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `rename_animation_library` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Property `reset_on_save` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Property `root_node` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `set_reset_on_save_enabled` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `seek` adds a new `update_only` optional parameter	✓	✓	✓	GH-80813
AnimationTree				
Method `_post_process_key_value` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Property `active` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `advance` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Signal `animation_finished` moved to base class `AnimationMixer`	✓	✗	✗	GH-80813
Signal `animation_started` moved to base class `AnimationMixer`	✓	✗	✗	GH-80813
Property `audio_max_polyphony` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `get_root_motion_position` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `get_root_motion_position_accumulator` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `get_root_motion_rotation` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `get_root_motion_rotation_accumulator` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `get_root_motion_scale` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Method `get_root_motion_scale_accumulator` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Property `process_callback` renamed to `callback_mode_process` and moved to base class `AnimationMixer`	✓	✓	✓	GH-80813
Property `root_motion_track` moved to base class `AnimationMixer`	✓	✓	✓	GH-80813

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Property `tree_root` changes type from `AnimationNode` to `AnimationRootNode`	✓	✗	✗	GH-80813

GUI nodes

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
PopupMenu				
Method `add_icon_shortcut` adds a new `allow_echo` optional parameter	✓	✓	✓	GH-36493
Method `add_shortcut` adds a new `allow_echo` optional parameter	✓	✓	✓	GH-36493
Method `clear` adds a new `free_submenus` optional parameter	✓	✓	✓	GH-79965
RichTextLabel				
Method `add_image` adds new `key`, `pad`, `tooltip`, and `size_in_percent` optional parameters	✓	✓	✓	GH-80410

Rendering

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
ImporterMesh				
Method `add_surface` changes `flags` parameter type from `uint32` to `uint64`	✓	✓	✓	GH-81138

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Method `get_surface_format` changes return type from `uint32` to `uint64`	✓	✗	✗	GH-81138
MeshDataTool				
Method `commit_to_surface` adds a new `compression_flags` optional parameter	✓	✓	✓	GH-81138
Method `get_format` changes return type from `uint32` to `uint64`	✓	✗	✗	GH-81138
RenderingDevice				
Enum field `BarrierMask.BARRIER_MASK_RASTER` changes value from `1` to `9`	✓	✓	✓	GH-79911
Enum field `BarrierMask.BARRIER_MASK_ALL_BARRIERS` changes value from `7` to `32767`	✓	✓	✓	GH-79911
Enum field `BarrierMask.BARRIER_MASK_NO_BARRIER` changes value from `8` to `32768`	✓	✓	✓	GH-79911
Method `shader_create_from_bytecode` adds a new `placeholder_rid` optional parameter	✓	✓	✓	GH-79606
Method `shader_get_vertex_input_attribute_ask` changes return type from `uint32` to `uint64`	✓	✗	✗	GH-81138
SurfaceTool				

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Method `commit` changes `flags` parameter type from `uint32` to `uint64`	✓	✓	✓	GH-81138

Text

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Font				
Method `set_fallbacks` replaced with `fallbacks` property	✓	✗	✗	GH-78266
Method `get_fallbacks` replaced with `fallbacks` property	✓	✗	✗	GH-78266
Method `find_variation` adds new `spacing_top`, `spacing_bottom`, `spacing_space`, and `spacing_glyph` optional parameters	✓	✓	✓	GH-80954

GraphEdit

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
GraphEdit				
Property `arrange_nodes_button_hidden` renamed to `show_arrange_button`	✗	✓	✓	GH-81582
Method `get_zoom_hbox` renamed to `get_menu_hbox`	✗	✓	✓	GH-79308

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Property `snap_distance` renamed to `snapping_distance`	✗	✓	✓	GH-79308
Property `use_snap` renamed to `snapping_enabled`	✗	✓	✓	GH-79308
GraphNode				
Property `comment` removed	✗	✗	✗	GH-79307
Signal `close_request` renamed to `delete_request` and moved to base class `GraphElement`	✗	✓	✓	GH-79311
Property `draggable` moved to base class `GraphElement`	✓	✓	✓	GH-79311
Property `draggable` moved to base class `GraphElement`	✓	✓	✓	GH-79311
Signal `dragged` moved to base class `GraphElement`	✓	✗	✗	GH-79311
Method `get_connection_input_color` removed	✗	✗	✗	GH-79311
Method `get_connection_input_count` removed	✗	✗	✗	GH-79311
Method `get_connection_input_height` removed	✗	✗	✗	GH-79311
Method `get_connection_input_position` removed	✗	✗	✗	GH-79311

Change	**GDScript Compatible**	**C# Binary Compatible**	**C# Source Compatible**	**Introduced**
Method `get_connection_input_slot` removed	✗	✗	✗	GH-79311
Method `get_connection_input_type` removed	✗	✗	✗	GH-79311
Method `get_connection_output_color` removed	✗	✗	✗	GH-79311
Method `get_connection_output_count` removed	✗	✗	✗	GH-79311
Method `get_connection_output_height` removed	✗	✗	✗	GH-79311
Method `get_connection_output_position` removed	✗	✗	✗	GH-79311
Method `get_connection_output_slot` removed	✗	✗	✗	GH-79311
Method `get_connection_output_type` removed	✗	✗	✗	GH-79311
Property `language` removed	✗	✗	✗	GH-79311
Signal `node_deselected` moved to base class `GraphElement`	✓	✓	✓	GH-79311
Signal `node_selected` moved to base class `GraphElement`	✓	✓	✓	GH-79311
Property `overlay` removed	✗	✗	✗	GH-79311

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Property `position_offset` moved to base class `GraphElement`	✓	✓	✓	GH-79311
Signal `position_offset_changed` moved to base class `GraphElement`	✓	✓	✓	GH-79311
Signal `raise_request` moved to base class `GraphElement`	✓	✓	✓	GH-79311
Property `resizable` moved to base class `GraphElement`	✓	✓	✓	GH-79311
Signal `resize_request` moved to base class `GraphElement`	✓	✗	✗	GH-79311
Property `selectable` moved to base class `GraphElement`	✓	✓	✓	GH-79311
Property `selected` moved to base class `GraphElement`	✓	✓	✓	GH-79311
Property `show_close` removed	✗	✗	✗	GH-79311
Property `text_direction` removed	✗	✗	✗	GH-79311

TileMap

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
TileMap				
Property `cell_quadrant_size` renamed to `rendering_quadrant_size`	✗	✓	✓	GH-81070

XR

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
XRInterface				
Property `environment_blend_mode` added	✓	✗	✗	GH-81561

Note

This change breaks compatibility in C# because the new property conflicts with the name of an existing enum and the C# bindings generator gives priority to properties, so the enum type was renamed from `EnvironmentBlendMode` to `EnvironmentBlendModeEnum`.

Upgrading from Godot 4.2 to Godot 4.3

For most games and apps made with 4.2 it should be relatively safe to migrate to 4.3. This page intends to cover everything you need to pay attention to when migrating your project.

Breaking changes

If you are migrating from 4.2 to 4.3, the breaking changes listed here might affect you. Changes are grouped by areas/systems.

This article indicates whether each breaking change affects GDScript and whether the C# breaking change is *binary compatible* or *source compatible*:

- **Binary compatible** - Existing binaries will load and execute successfully without recompilation, and the run-time behavior won't change.
- **Source compatible** - Source code will compile successfully without changes when upgrading Godot.

GDExtension

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
GDExtension				
Method `close_library` removed	❌	❌	❌	GH-88418
Method `initialize_library` removed	❌	❌	❌	GH-88418
Method `open_library` removed	❌	❌	❌	GH-88418

Since it was basically impossible to use these methods in any useful way, these methods have been removed. Use `GDExtensionManager::load_extension` and `GDExtensionManager::unload_extension` instead to correctly load and unload a GDExtension.

Animation

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Animation				

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Method `position_track_interpolate` adds a new `backward` optional parameter	✓	✓	✓	GH-86629
Method `rotation_track_interpolate` adds a new `backward` optional parameter	✓	✓	✓	GH-86629
Method `scale_track_interpolate` adds a new `backward` optional parameter	✓	✓	✓	GH-86629
Method `blend_shape_track_interpolate` adds a new `backward` optional parameter	✓	✓	✓	GH-86629
Method `value_track_interpolate` adds a new `backward` optional parameter	✓	✓	✓	GH-86629
Method `track_find_key` adds a new `limit` optional parameter	✓	✓	✓	GH-86661
Method `track_find_key` adds a new `backward` optional parameter	✓	✓	✓	GH-92861
AnimationMixer				
Method `_post_process_key_value` changes `object` parameter type from `Object` to `uint64`	✓	✗	✗	GH-86687
Skeleton3D				
Method `add_bone` changes return type from `void` to `int32`	✓	✗	✓	GH-88791
Signal `bone_pose_changed` replaced by `skeleton_updated`	✗	✗	✗	GH-90575
BoneAttachment3D				
Method `on_bone_pose_update` replaced by `on_skeleton_update`	✓	✓	✓	GH-90575

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced

GUI nodes

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
AcceptDialog				
Method `register_text_enter` changes parameter `line_edit` type from `Control` to `LineEdit`	✓	✓	✓	[GH-89419](https://github.com/godotengine/godot/pull/89419)
Method `remove_button` changes parameter `button` type from `Control` to `Button`	✓	✓	✓	[GH-89419](https://github.com/godotengine/godot/pull/89419)

Physics

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
PhysicsShapeQueryParameters3D				
Property `motion` changes type from `Vector2` to `Vector3`	✗	✗	✗	[GH-85393](https://github.com/godotengine/godot/pull/85393)

Note

In C#, the enum `PhysicsServer3D.G6DofJointAxisFlag` breaks compatibility because of the way the bindings generator detects the enum prefix. New members were added in GH-89851 [https://github.com/godotengine/godot/pull/89851] to the enum that caused the enum members to be renamed.

Rendering

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Intro
RenderingDevice				

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Intro
Enum field `FinalAction.FINAL_ACTION_CONTINUE` changes value from `2` to `0`	✓	✗	✗	GH-84
Enum field `InitialAction.INITIAL_ACTION_CLEAR` changes value from `0` to `1`	✓	✗	✗	GH-84
Enum field `InitialAction.INITIAL_ACTION_CLEAR_REGION_CONTINUE` changes value from `2` to `1`	✓	✗	✗	GH-84
Enum field `InitialAction.INITIAL_ACTION_CONTINUE` changes value from `5` to `0`	✓	✗	✗	GH-84
Enum field `InitialAction.INITIAL_ACTION_DROP` changes value from `4` to `2`	✓	✗	✗	GH-84
Enum field `InitialAction.INITIAL_ACTION_KEEP` changes value from `3` to `0`	✓	✗	✗	GH-84
Method `buffer_clear` removes `post_barrier` parameter	✓	✓	✓	GH-84
Method `buffer_update` removes `post_barrier` parameter	✓	✓	✓	GH-84
Method `compute_list_begin` removes `allow_draw_overlap` parameter	✓	✓	✓	GH-84
Method `compute_list_end` removes `post_barrier` parameter	✓	✓	✓	GH-84
Method `draw_list_begin` removes `storage_textures` parameter	✓	✓	✓	GH-84
Method `draw_list_end` removes `post_barrier` parameter	✓	✓	✓	GH-84

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Intro
Method `texture_clear` removes `post_barrier` parameter	✓	✓	✓	GH-84
Method `texture_copy` removes `post_barrier` parameter	✓	✓	✓	GH-84
Method `texture_resolve_multisample` removes `post_barrier` parameter	✓	✓	✓	GH-84
Method `texture_update` removes `post_barrier` parameter	✓	✓	✓	GH-84
RenderingServer				
Method `environment_set_fog` adds a new `fog_mode` optional parameter	✓	✓	✓	GH-84
RenderSceneBuffersRD				
Method `get_color_layer` adds a new `msaa` optional parameter	✓	✓	✓	GH-80
Method `get_depth_layer` adds a new `msaa` optional parameter	✓	✓	✓	GH-80
Method `get_velocity_layer` adds a new `msaa` optional parameter	✓	✓	✓	GH-80
Method `get_color_texture` adds a new `msaa` optional parameter	✓	✓	✓	GH-80
Method `get_depth_texture` adds a new `msaa` optional parameter	✓	✓	✓	GH-80
Method `get_velocity_texture` adds a new `msaa` optional parameter	✓	✓	✓	GH-80

Note

While the values of the enum fields in `RenderingDevice.InitialAction` and `RenderingDevice.FinalAction` changed, the only method that consumed them (`draw_list_begin`) added a compatibility method which supports the old values. So in practice it doesn't break compatibility.

Note

In C#, the enum `RenderingDevice.DriverResource` breaks compatibility because of the way the bindings generator detects the enum prefix. New members were added in GH-83452 [https://github.com/godotengine/godot/pull/83452] to the enum that caused the enum members to be renamed.

Text

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Font				
Method `find_variation` adds a new `baseline_offset` optional parameter	✓	✓	✓	GH-87668
RichTextLabel				
Method `push_meta` adds a new `underline_mode` optional parameter	✓	✓	✓	GH-89024
TextServer				
Method `shaped_text_get_word_breaks` adds a new optional `skip_grapheme_flags` parameter	✓	✓	✓	GH-90732
TextServerExtension				
Method `_shaped_text_get_word_breaks` adds a new `skip_grapheme_flags` parameter	✗	✗	✗	GH-90732

Audio

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
AudioStreamPlaybackPolyphonic				
Method `play_stream` adds new `playback_type`, and `bus` optional parameters	✓	✓	✓	GH-91382

Navigation

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
AStar2D				
Method `get_id_path` adds new `allow_partial_path` optional parameter	✓	✓	✓	GH-88047
Method `get_point_path` adds new `allow_partial_path` optional parameter	✓	✓	✓	GH-88047
AStar3D				
Method `get_id_path` adds new `allow_partial_path` optional parameter	✓	✓	✓	GH-88047
Method `get_point_path` adds new `allow_partial_path` optional parameter	✓	✓	✓	GH-88047
AStarGrid2D				
Method `get_id_path` adds new `allow_partial_path` optional parameter	✓	✓	✓	GH-88047
Method `get_point_path` adds new `allow_partial_path` optional parameter	✓	✓	✓	GH-88047
NavigationRegion2D				

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
Property `avoidance_layers` removed	❌	❌	❌	GH-90747
Property `constrain_avoidance` removed	❌	❌	❌	GH-90747
Method `get_avoidance_layer_value` removed	❌	❌	❌	GH-90747
Method `set_avoidance_layer_value` removed	❌	❌	❌	GH-90747

Note

The constrain avoidance feature in `NavigationRegion2D` was experimental and has been discontinued with no replacement.

TileMap

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
TileData				
Method `get_navigation_polygon` adds new `flip_h`, `flip_v`, and `transpose` optional parameters	✓	✓	✓	GH-84660
Method `get_occluder` adds new `flip_h`, `flip_v`, and `transpose` optional parameters	✓	✓	✓	GH-84660

XR

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
WebXRInterface				
Method `get_input_source_tracker` changes return type from `XRPositionalTracker` to `XRControllerTracker`	✓	❌	✓	GH-90645

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
XRServer				
Method `get_tracker` changes return type from `XRPositionalTracker` to `XRTracker`	✓	✗	✗	GH-90645

Editor plugins

Change	GDScript Compatible	C# Binary Compatible	C# Source Compatible	Introduced
EditorInspectorPlugin				
Method `add_property_editor` adds a new `label` optional parameter	✓	✓	✓	GH-92322
EditorPlugin				
Method `add_control_to_bottom_panel` adds a new `shortcut` optional parameter	✓	✓	✓	GH-88081
Method `add_control_to_dock` adds a new `shortcut` optional parameter	✓	✓	✓	GH-88081
EditorSceneFormatImporterFBX				
Type renamed to `EditorSceneFormatImporterFBX2GLTF`	✗	✗	✗	GH-81746

Behavior changes

In 4.3 some behavior changes have been introduced, which might require you to adjust your project.

Core

Note

Binary serialization was modified to fix some issues with the serialization of scripted Objects and typed Arrays (GH-78219 [https://github.com/godotengine/godot/pull/78219]). This breaks compat with script encoding/decoding.

Note

`PackedByteArray` is now able to use a more compact base64 encoding for storage. But the trade-off is that it breaks compatibility, meaning that older versions of Godot may not be able to open resources saved by 4.3 (GH-89186 [https://github.com/godotengine/godot/pull/89186]).

To maximize compatibility, this new storage format will only be enabled for resources and scenes that contain large PackedByteArrays for now. Support for this new format will also be added in patch updates for older versions of Godot. Once all supported Godot versions are able to read the new format, we will gradually retire the compatibility measures and have all resources and scenes use the new storage format.

Note

In C#, the `Transform3D.InterpolateWith` implementation was fixed to use the right order of operations, applying the rotation before the scale (GH-89843 [https://github.com/godotengine/godot/pull/89843]).

Note

In C#, the `Aabb.GetSupport` implementation was fixed to properly return the support vector (GH-88919 [https://github.com/godotengine/godot/pull/88919]).

Note

In C#, the Variant types' `ToString` implementation now defaults to using the `InvariantCulture` (GH-89547 [https://github.com/godotengine/godot/pull/89547]) which means `Vector2(1.2, 3.4)` is formatted using `.` as the decimal separator independently of the language of the operating system that the program is running on.

Animation

Note

`AnimationMixer` replaced its Capture mode with a new Capture feature that works much better than the old one, this replaces the existing cache (GH-86715 [https://github.com/godotengine/godot/pull/86715]).

Note

`AnimationNode` has a reworked process for retrieving the semantic time info. This ensures that time-related behavior works as expected, but changes the blending behavior. Implementors of the `_process` virtual method should also note that this method is now deprecated and will be replaced by a new one in the future (GH-87171 [https://github.com/godotengine/godot/pull/87171]).

More information about the changes to Animation can be found in the Migrating Animations from Godot 4.0 to 4.3 [https://godotengine.org/article/migrating-animations-from-godot-4-0-to-4-3] article.

GUI nodes

Note

The default font outline color was changed from white to black (GH-54641 [https://github.com/godotengine/godot/pull/54641]).

Note

The `auto_translate` property is deprecated in favor of the `auto_translate_mode` property which is now in `Node` (GH-87530 [https://github.com/godotengine/godot/pull/87530]). The default value for `auto_translate_mode` is `AUTO_TRANSLATE_INHERIT`, which means nodes inherit the `auto_translate_mode` value from their parent. This means, existing nodes with the `auto_translate` property set to `true` may no longer be translated if they are children of a node with the `auto_translate` property set to `false`.

Multiplayer

Note

The `SceneMultiplayer` caching protocol was changed to send the received ID instead of the Node path when sending a node removal confirmation packet (GH-90027 [https://github.com/godotengine/godot/pull/90027]).

This is a breaking change for the high-level multiplayer protocol making it incompatible with previous Godot versions. Upgrade both your server and client versions to Godot 4.3 to handle this change gracefully.

Note that high-level multiplayer facilities are only ever meant to be compatible with server and client using the same Godot version. It is recommended to implement some kind of version checking.

Rendering

Note

Decals now convert the modulate color from an sRGB color to a linear color, like all other inputs, to ensure proper blending (GH-89849 [https://github.com/godotengine/godot/pull/89849]). Existing projects that were using the decal's modulate property will notice a change in their visuals.

Note

The reverse Z depth buffer technique is now implemented. This may break compatibility for some shaders. Read the Introducing Reverse Z (AKA I'm sorry for breaking your shader) [https://godotengine.org/article/introducing-reverse-z/] article for more information and guidance on how to fix common scenarios.

TileMap

Note

`TileMap` layers were moved to individual nodes (GH-87379 [https://github.com/godotengine/godot/pull/87379] and GH-89179 [https://github.com/godotengine/godot/pull/89179]).

Android

Note

Android permissions are no longer requested automatically because it goes against the recommended best practices (GH-87080 [https://github.com/godotengine/godot/pull/87080]). Use the `request_permission` method in `OS` and the `on_request_permissions_result` signal on `MainLoop` to request permissions and wait for the user response.

2D

- Canvas layers
- Viewport and canvas transforms

Rendering

- 2D lights and shadows
- 2D meshes
- 2D sprite animation
- 2D particle systems
- 2D antialiasing
- Custom drawing in 2D
- 2D Parallax

Physics and movement

- 2D movement overview

Tools

- Using TileSets
- Using TileMaps

Canvas layers

Viewport and Canvas items

CanvasItem is the base for all 2D nodes, be it regular 2D nodes, such as Node2D, or Control. Both inherit from CanvasItem. You can arrange canvas items in trees. Each item will inherit its parent's transform: when the parent moves, its children move too.

CanvasItem nodes, and nodes inheriting from them, are direct or indirect children of a Viewport, that displays them.

The Viewport's property Viewport.canvas_transform, allows to apply a custom Transform2D transform to the CanvasItem hierarchy it contains. Nodes such as Camera2D work by changing that transform.

To achieve effects like scrolling, manipulating the canvas transform property is more efficient than moving the root canvas item and the entire scene with it.

Usually though, we don't want *everything* in the game or app to be subject to the canvas transform. For example:

- **Parallax Backgrounds**: Backgrounds that move slower than the rest of the stage.
- **UI**: Think of a user interface (UI) or head-up display (HUD) superimposed on our view of the game world. We want a life counter, score display and other elements to retain their screen positions even when our view of the game world changes.
- **Transitions**: We may want visual effects used for transitions (fades, blends) to remain at a fixed screen location.

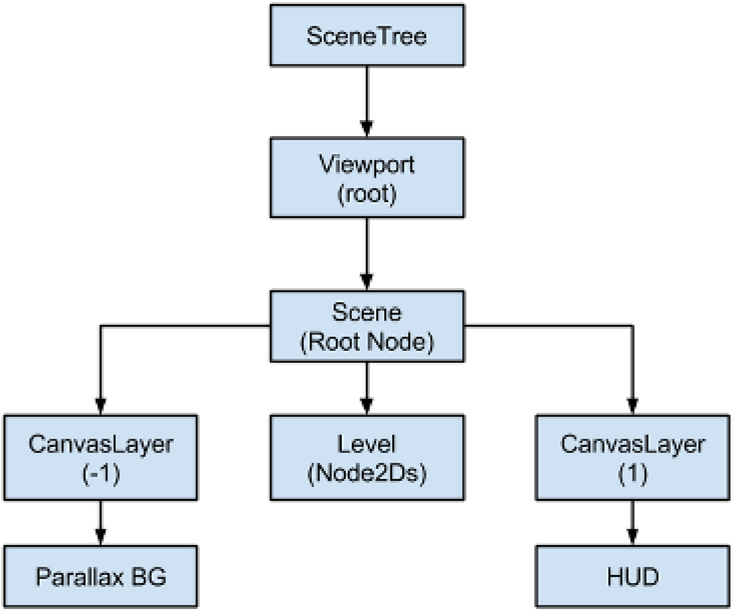
How to solve these problems in a single scene tree?

CanvasLayers

The answer is CanvasLayer, which is a node that adds a separate 2D rendering layer for all its children and grand-children. Viewport children will draw by default at layer "0", while a CanvasLayer will draw at any numeric layer. Layers with a greater number will be drawn above those with a smaller number. CanvasLayers also have their own transform and do not depend on the transform of other layers. This allows the UI to be fixed in screen-space while our view on the game world changes.

An example of this is creating a parallax background. This can be done with a CanvasLayer at layer "-1". The screen with the points, life counter and pause button can also be created at layer "1".

Here's a diagram of how it looks:


SceneTree
Viewport (root)
Scene (Root Node)
CanvasLayer (-1)
Level (Node2Ds)
CanvasLayer (1)
Parallax BG
HUD

CanvasLayers are independent of tree order, and they only depend on their layer number, so they can be instantiated when needed.

Note

CanvasLayers aren't necessary to control the drawing order of nodes. The standard way to ensuring that a node is correctly drawn 'in front' or 'behind' others is to manipulate the order of the nodes in the scene panel. Perhaps counterintuitively, the topmost nodes in the scene panel are drawn on *behind* lower ones in the viewport. 2D nodes also have the CanvasItem.z_index property for controlling their drawing order.

Viewport and canvas transforms

Introduction

This is an overview of the 2D transforms going on for nodes from the moment they draw their content locally to the time they are drawn onto the screen. This overview discusses very low level details of the engine.

The goal of this tutorial is to teach a way for feeding input events to the Input with a position in the correct coordinate system.

A more extensive description of all coordinate systems and 2d transforms is available in 2D coordinate systems and 2D transforms.

Canvas transform

As mentioned in the previous tutorial, Canvas layers, every CanvasItem node (remember that Node2D and Control based nodes use CanvasItem as their common root) will reside in a *Canvas Layer*. Every canvas layer has a transform (translation, rotation, scale, etc.) that can be accessed as a Transform2D.

Also covered in the previous tutorial, nodes are drawn by default in Layer 0, in the built-in canvas. To put nodes in a different layer, a CanvasLayer node can be used.

Global canvas transform

Viewports also have a Global Canvas transform (also a Transform2D). This is the master transform and affects all individual *Canvas Layer* transforms. Generally, this is primarily used in Godot's CanvasItem Editor.

Stretch transform

Finally, viewports have a *Stretch Transform*, which is used when resizing or stretching the screen. This transform is used internally (as described in Multiple resolutions), but can also be manually set on each viewport.

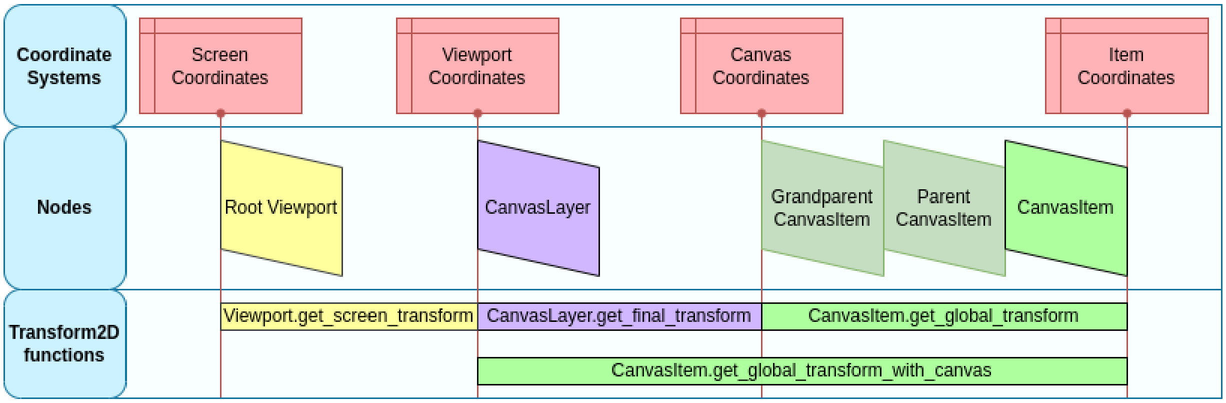
Input events are multiplied by this transform but lack the ones above. To convert InputEvent coordinates to local CanvasItem coordinates, the CanvasItem.make_input_local() function was added for convenience.

Window transform

The root viewport is a Window. In order to scale and position the *Window's* content as described in Multiple resolutions, each Window contains a *window transform*. It is for example responsible for the black bars at the *Window's* sides so that the *Viewport* is displayed with a fixed aspect ratio.

Transform order

To convert a CanvasItem local coordinate to an actual screen coordinate, the following chain of transforms must be applied:


Coordinate Systems
Screen Coordinates
Viewport Coordinates
Canvas Coordinates
Item Coordinates
Nodes
Root Viewport
CanvasLayer
Grandparent CanvasItem
Parent CanvasItem
CanvasItem
Transform2D functions
Viewport.get_screen_transform
CanvasLayer.get_final_transform
CanvasItem.get_global_transform
CanvasItem.get_global_transform_with_canvas

Transform functions

The above graphic shows some available transform functions. All transforms are directed from right to left, this means multiplying a transform with a coordinate results in a coordinate system further to the left, multiplying the affine inverse of a transform results in a coordinate system further to the right:

GDScript

```
# Called from a CanvasItem.
canvas_pos = get_global_transform() * local_pos
local_pos = get_global_transform().affine_inverse() *
canvas_pos
```

C#

```
// Called from a CanvasItem.
canvasPos = GetGlobalTransform() * localPos;
localPos = GetGlobalTransform().AffineInverse() * canvasPos;
```

Finally, then, to convert a CanvasItem local coordinates to screen coordinates, just multiply in the following order:

GDScript

```
var screen_coord = get_viewport().get_screen_transform() *
```

```
get_global_transform_with_canvas() * local_pos
```

C#

```
var screenCoord = GetViewport().GetScreenTransform() *
GetGlobalTransformWithCanvas() * localPos;
```

Keep in mind, however, that it is generally not desired to work with screen coordinates. The recommended approach is to simply work in Canvas coordinates (`CanvasItem.get_global_transform()`), to allow automatic screen resolution resizing to work properly.

Feeding custom input events

It is often desired to feed custom input events to the game. With the above knowledge, to correctly do this in the focused window, it must be done the following way:

GDScript

```
var local_pos = Vector2(10, 20) # Local to Control/Node2D.
var ie = InputEventMouseButton.new()
ie.button_index = MOUSE_BUTTON_LEFT
ie.position = get_viewport().get_screen_transform() *
get_global_transform_with_canvas() * local_pos
Input.parse_input_event(ie)
```

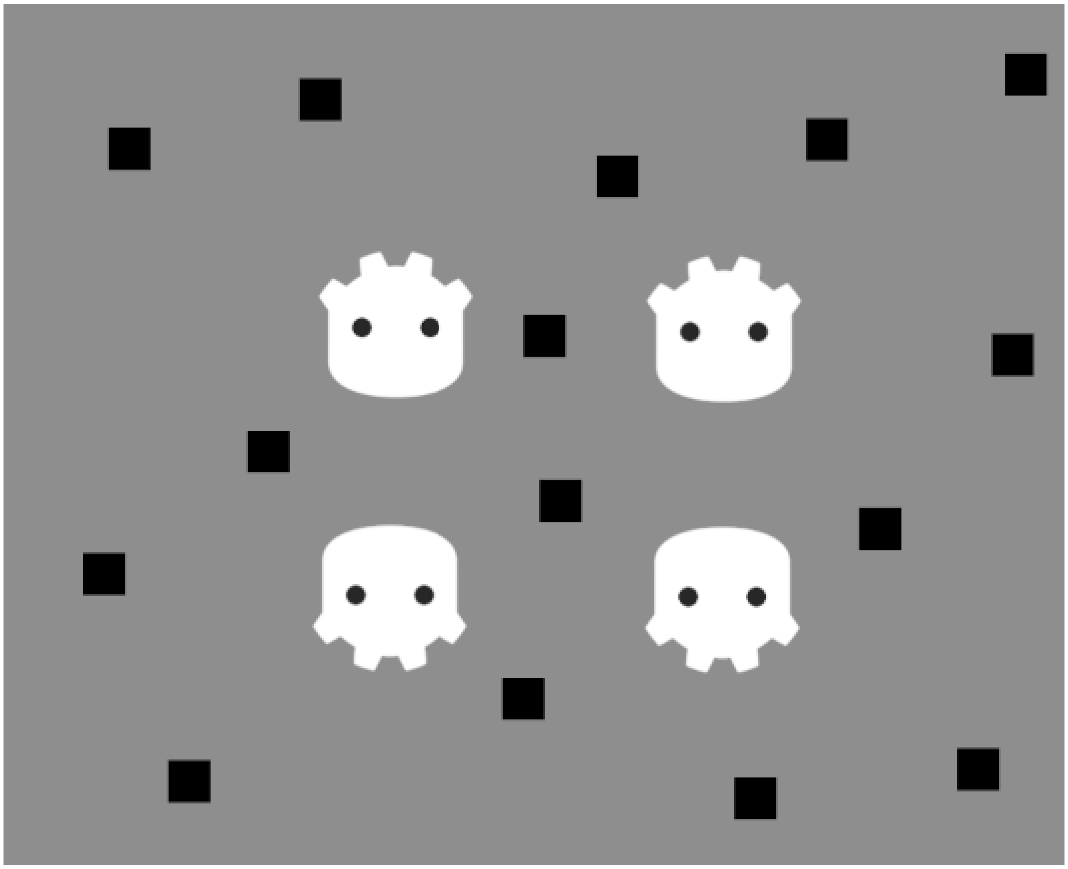
C#

```
var localPos = new Vector2(10,20); // Local to Control/Node2D.
var ie = new InputEventMouseButton()
{
    ButtonIndex = MouseButton.Left,
    Position = GetViewport().GetScreenTransform() *
GetGlobalTransformWithCanvas() * localPos,
};
Input.ParseInputEvent(ie);
```

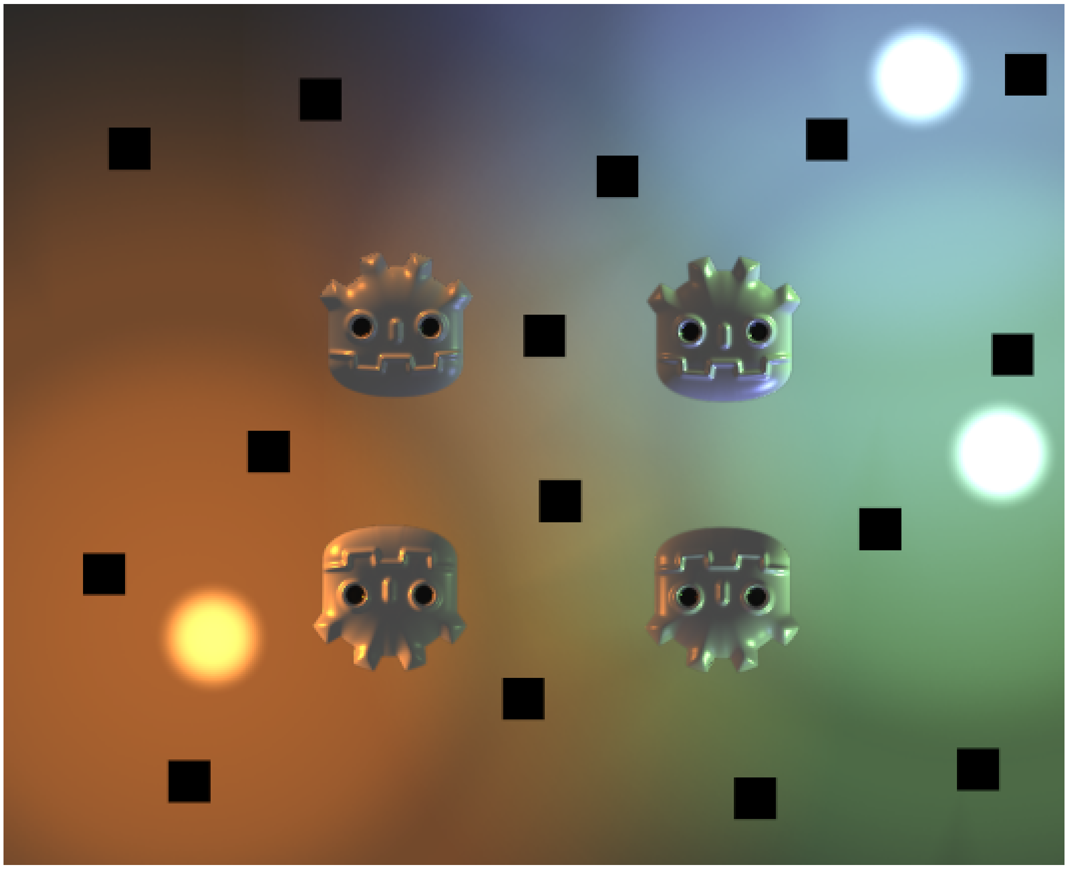
2D lights and shadows

Introduction

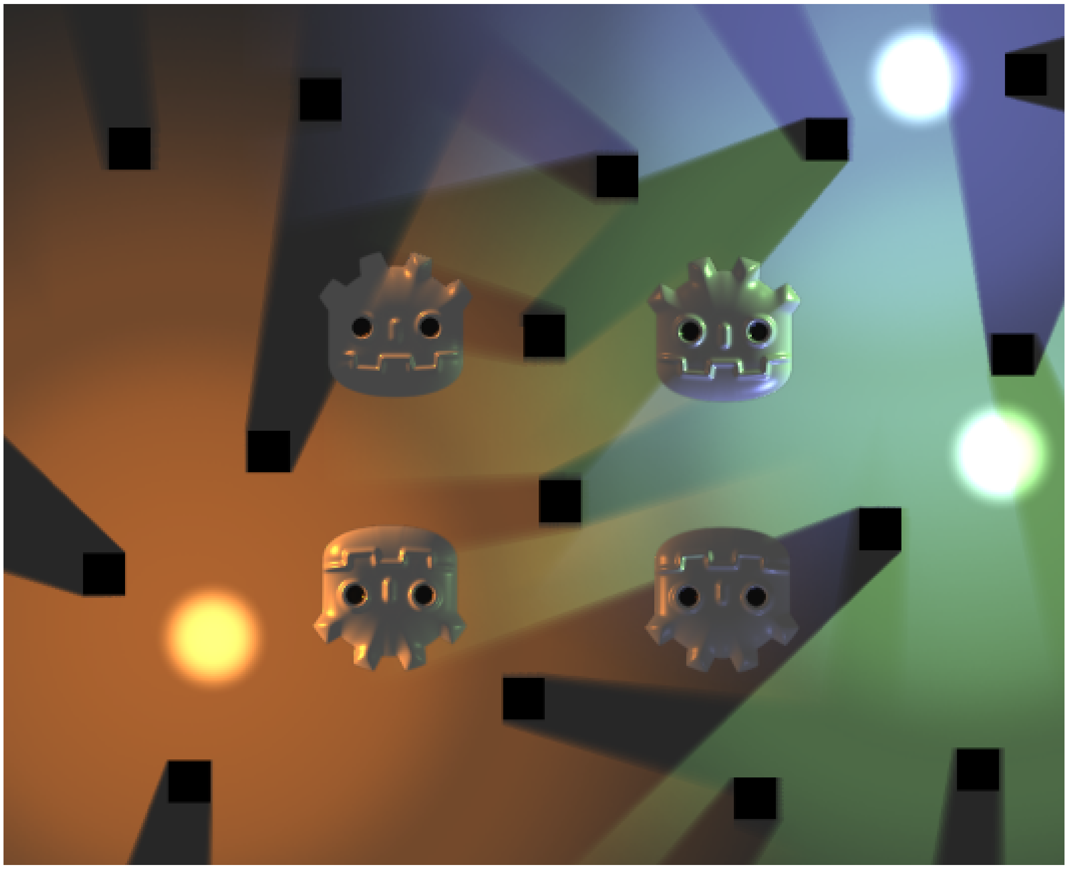
By default, 2D scenes in Godot are unshaded, with no lights and shadows visible. While this is fast to render, unshaded scenes can look bland. Godot provides the ability to use real-time 2D lighting and shadows, which can greatly enhance the sense of depth in your project.



No 2D lights or shadows, scene is unshaded



2D lights enabled (without shadows)



2D lights and shadows enabled

Nodes

There are several nodes involved in a complete 2D lighting setup:

- CanvasModulate (to darken the rest of the scene)
- PointLight2D (for omnidirectional or spot lights)
- DirectionalLight2D (for sunlight or moonlight)
- LightOccluder2D (for light shadow casters)
- Other 2D nodes that receive lighting, such as Sprite2D or TileMapLayer.

CanvasModulate is used to darken the scene by specifying a color that will act as the base "ambient" color. This is the final lighting color in areas that are *not* reached by any 2D light. Without a

CanvasModulate node, the final scene would look too bright as 2D lights would only brighten the existing unshaded appearance (which appears fully lit).

Sprite2Ds are used to display the textures for the light blobs, the background, and for the shadow casters.

PointLight2Ds are used to light the scene. The way a light typically works is by adding a selected texture over the rest of the scene to simulate lighting.

LightOccluder2Ds are used to tell the shader which parts of the scene cast shadows. These occluders can be placed as independent nodes or can be part of a TileMapLayer node.

The shadows appear only on areas covered by the PointLight2D and their direction is based on the center of the Light.

Note

The background color does **not** receive any lighting. If you want light to be cast on the background, you need to add a visual representation for the background, such as a Sprite2D.

The Sprite2D's **Region** properties can be helpful to quickly create a repeating background texture, but remember to also set **Texture > Repeat** to **Enabled** in the Sprite2D's properties.

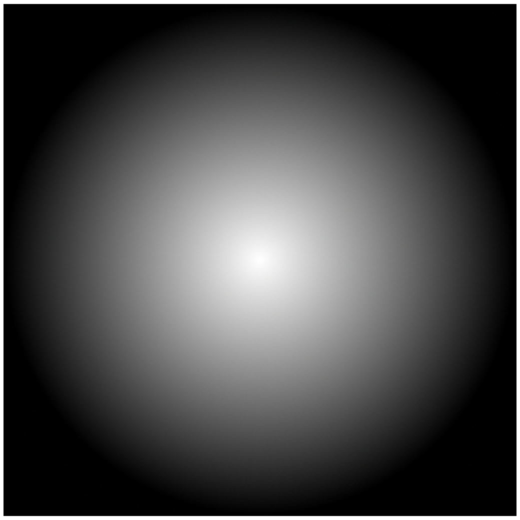
Point lights

Point lights (also called positional lights) are the most common element in 2D lighting. Point lights can be used to represent light from torches, fire, projectiles, etc.

PointLight2D offers the following properties to tweak in the inspector:

- **Texture:** The texture to use as a light source. The texture's size determines the size of the light. The texture may have an alpha channel, which is useful when using Light2D's **Mix** blend mode, but it is not required if using the **Add** (default) or **Subtract** blend modes.
- **Offset:** The offset for the light texture. Unlike when you move the light node, changing the offset does *not* cause shadows to move.
- **Texture Scale:** The multiplier for the light's size. Higher values will make the light extend out further. Larger lights have a higher performance cost as they affect more pixels on screen, so consider this before increasing a light's size.
- **Height:** The light's virtual height with regards to normal mapping. By default, the light is very close to surfaces receiving lights. This will make lighting hardly visible if normal mapping is used, so consider increasing this value. Adjusting the light's height only makes a visible difference on surfaces that use normal mapping.

If you don't have a pre-made texture to use in a light, you can use this "neutral" point light texture (right-click > **Save Image As...**):



Neutral point light texture

If you need different falloff, you can procedurally create a texture by assigning a **New GradientTexture2D** on the light's **Texture** property. After creating the resource, expand its **Fill** section and set the fill mode to **Radial**. You will then have to adjust the gradient itself to start from opaque white to transparent white, and move its starting location to be in the center.

Directional light

New in Godot 4.0 is the ability to have directional lighting in 2D. Directional lighting is used to represent sunlight or moonlight. Light rays are casted parallel to each other, as if the sun or moon was infinitely far away from the surface that is receiving the light.

DirectionalLight2D offers the following properties:

- **Height:** The light's virtual height with regards to normal mapping (`0.0` = parallel to surfaces, `1.0` = perpendicular to surfaces). By default, the light is fully parallel with the surfaces receiving lights. This will make lighting hardly visible if normal mapping is used, so consider increasing this value. Adjusting the light's height only makes a visual difference on surfaces that use normal mapping. **Height** does not affect shadows' appearance.
- **Max Distance:** The maximum distance from the camera center objects can be before their shadows are culled (in pixels). Decreasing this value can prevent objects located outside the camera from casting shadows (while also improving performance). Camera2D zoom is not taken into account by **Max Distance**, which means that at higher zoom values, shadows will appear to fade out sooner when zooming onto a given point.

Note

Directional shadows will always appear to be infinitely long, regardless of the value of the **Height** property. This is a limitation of the shadow rendering method used for 2D lights in Godot.

To have directional shadows that are not infinitely long, you should disable shadows in the DirectionalLight2D and use a custom shader that reads from the 2D signed distance field instead. This distance field is automatically generated from LightOccluder2D nodes present in the scene.

Common light properties

Both PointLight2D and DirectionalLight2D offer common properties, which are part of the Light2D base class:

- **Enabled:** Allows toggling the light's visibility. Unlike hiding the light node, disabling this property will not hide the light's children.
- **Editor Only:** If enabled, the light is only visible within the editor. It will be automatically disabled in the running project.
- **Color:** The light's color.
- **Energy:** The light's intensity multiplier. Higher values result in a brighter light.
- **Blend Mode:** The blending formula used for light computations. The default **Add** is suited for most use cases. **Subtract** can be used for negative lights, which are not physically accurate but can be used for special effects. The **Mix** blend mode mixes the value of pixels corresponding to the light's texture with the values of pixels under it by linear interpolation.
- **Range > Z Min:** The lowest Z index affected by the light.
- **Range > Z Max:** The highest Z index affected by the light.
- **Range > Layer Min:** The lowest visual layer affected by the light.

- **Range > Layer Max:** The highest visual layer affected by the light.
- **Range > Item Cull Mask:** Controls which nodes receive light from this node, depending on the other nodes' enabled visual layers **Occluder Light Mask**. This can be used to prevent certain objects from receiving light.

Setting up shadows

After enabling the **Shadow > Enabled** property on a PointLight2D or DirectionalLight2D node, you will not see any visual difference initially. This is because no nodes in your scene have any *occluders* yet, which are used as a basis for shadow casting.

For shadows to appear in the scene, LightOccluder2D nodes must be added to the scene. These nodes must also have occluder polygons that are designed to match the sprite's outline.

Along with their polygon resource (which must be set to have any visual effect), LightOccluder2D nodes have 2 properties:

- **SDF Collision:** If enabled, the occluder will be part of a real-time generated *signed distance field* that can be used in custom shaders. When not using custom shaders that read from this SDF, enabling this makes no visual difference and has no performance cost, so this is enabled by default for convenience.
- **Occluder Light Mask:** This is used in tandem with PointLight2D and DirectionalLight2D's **Shadow > Item Cull Mask** property to control which objects cast shadows for each light. This can be used to prevent specific objects from casting shadows.

There are two ways to create light occluders:

Automatically generating a light occluder

Occluders can be created automatically from Sprite2D nodes by selecting the node, clicking the **Sprite2D** menu at the top of the 2D editor then choosing **Create LightOccluder2D Sibling**.

In the dialog that appears, an outline will surround your sprite's edges. If the outline matches the sprite's edges closely, you can click **OK**. If the outline is too far away from the sprite's edges (or is "eating" into the sprite's edges), adjust **Grow (pixels)** and **Shrink (pixels)**, then click **Update Preview**. Repeat this operation until you get satisfactory results.

Manually drawing a light occluder

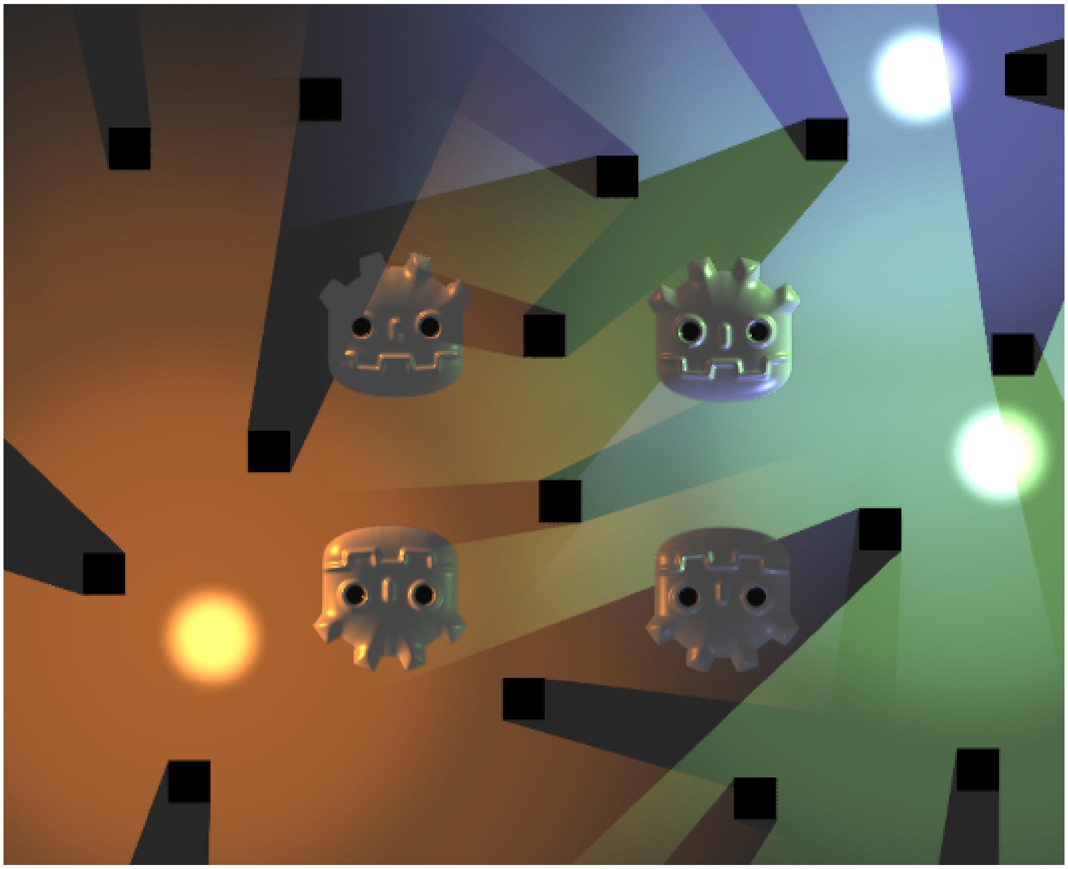
Create a LightOccluder2D node, then select the node and click the "+" button at the top of the 2D editor. When asked to create a polygon resource, answer **Yes**. You can then start drawing an occluder polygon by clicking to create new points. You can remove existing points by right-clicking them, and you can create new points from the existing line by clicking on the line then dragging.

The following properties can be adjusted on 2D lights that have shadows enabled:

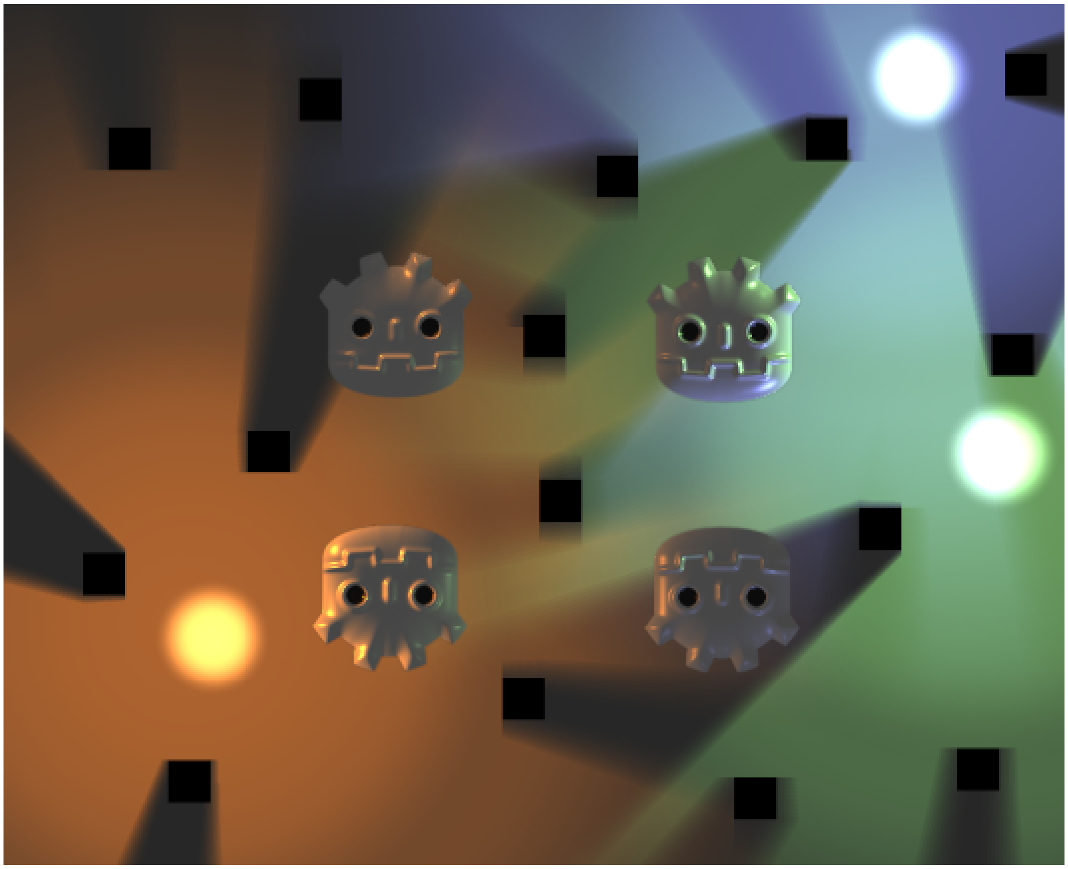
- **Color:** The color of shaded areas. By default, shaded areas are fully black, but this can be changed for artistic purposes. The color's alpha channel controls how much the shadow is tinted by the specified color.
- **Filter:** The filter mode to use for shadows. The default **None** is the fastest to render, and is well suited for games with a pixel art aesthetic (due to its "blocky" visuals). If you want a soft shadow, use **PCF5** instead. **PCF13** is even softer, but is the most demanding to render. PCF13 should only be used for a few lights at once due to its high rendering cost.
- **Filter Smooth:** Controls how much softening is applied to shadows when **Filter** is set to **PCF5** or **PCF13**. Higher values

result in a softer shadow, but may cause banding artifacts to be visible (especially with PCF5).

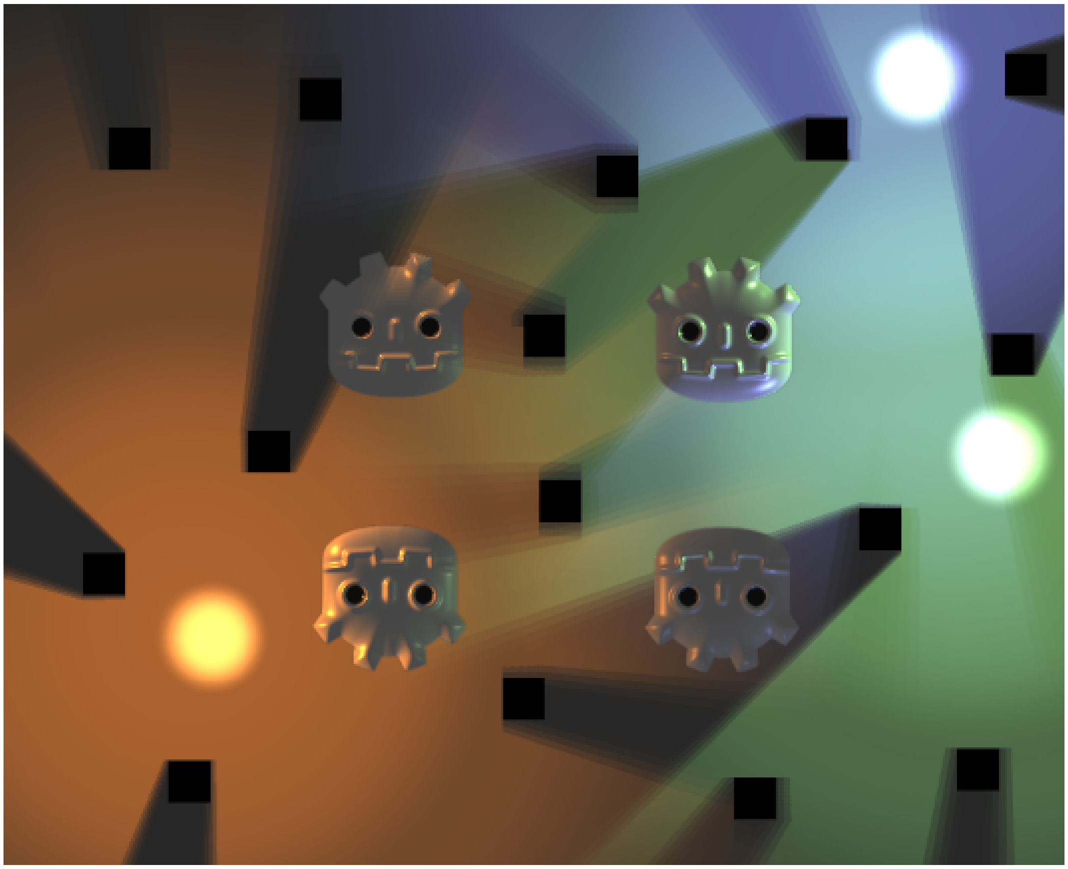
- **Item Cull Mask:** Controls which LightOccluder2D nodes cast shadows, depending on their respective **Occluder Light Mask** properties.



Hard shadows



Soft shadows (PCF13, Filter Smooth 1.5)



Soft shadows with streaking artifacts due to Filter Smooth being too high (PCF5, Filter Smooth 4)

Occluder draw order

LightOccluder2Ds follows the usual 2D drawing order. This is important for 2D lighting, as this is how you control whether the occluder should occlude the sprite itself or not.

If the LightOccluder2D node is a *sibling* of the sprite, the occluder will occlude the sprite itself if it's placed *below* the sprite in the scene tree.

If the LightOccluder2D node is a *child* of the sprite, the occluder will occlude the sprite itself if **Show Behind Parent** is disabled on the LightOccluder2D node (which is the default).

Normal and specular maps

Normal maps and specular maps can greatly enhance the sense of depth of your 2D lighting. Similar to how these work in 3D rendering, normal maps can help make lighting look less flat by varying its intensity depending on the direction of the surface receiving light (on a per-pixel basis). Specular maps further help improve visuals by making some of the light reflect back to the viewer.

Both PointLight2D and DirectionalLight2D support normal mapping and specular mapping. Since Godot 4.0, normal and specular maps can be assigned to any 2D element, including nodes that inherit from Node2D or Control.

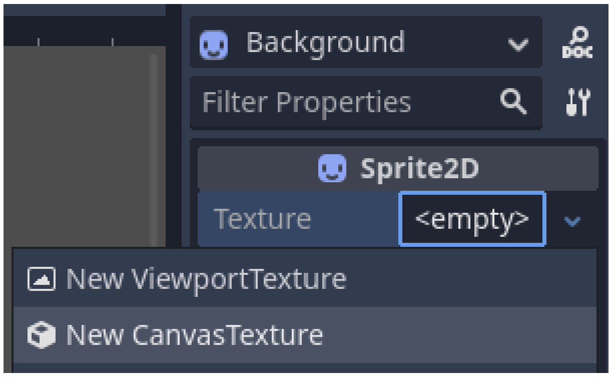
A normal map represents the direction in which each pixel is "pointing" towards. This information is then used by the engine to correctly apply lighting to 2D surfaces in a physically plausible way. Normal maps are typically created from hand-painted height maps, but they can also be automatically generated from other textures.

A specular map defines how much each pixel should reflect light (and in which color, if the specular map contains color). Brighter values will result in a brighter reflection at that given spot on the texture. Specular maps are typically created with manual editing, using the diffuse texture as a base.

Tip

If you don't have normal or specular maps for your sprites, you can generate them using the free and open source Laigter [https://azagaya.itch.io/laigter] tool.

To set up normal maps and/or specular maps on a 2D node, create a new CanvasTexture resource for the property that draws the node's texture. For example, on a Sprite2D:



Creating a CanvasTexture resource for a Sprite2D node

Expand the newly created resource. You can find several properties you will need to adjust:

- **Diffuse > Texture:** The base color texture. In this property, load the texture you're using for the sprite itself.
- **Normal Map > Texture:** The normal map texture. In this property, load a normal map texture you've generated from a height map (see the tip above).
- **Specular > Texture:** The specular map texture, which controls the specular intensity of each pixel on the diffuse texture. The specular map is usually grayscale, but it can also contain color to multiply the color of reflections accordingly. In this property, load a specular map texture you've created (see the tip above).
- **Specular > Color:** The color multiplier for specular reflections.
- **Specular > Shininess:** The specular exponent to use for reflections. Lower values will increase the brightness of reflections and make them more diffuse, while higher values will make reflections more localized. High values are more suited for wet-looking surfaces.

- **Texture > Filter:** Can be set to override the texture filtering mode, regardless of what the node's property is set to (or the **Rendering > Textures > Canvas Textures > Default Texture Filter** project setting).
- **Texture > Repeat:** Can be set to override the texture filtering mode, regardless of what the node's property is set to (or the **Rendering > Textures > Canvas Textures > Default Texture Repeat** project setting).

After enabling normal mapping, you may notice that your lights appear to be weaker. To resolve this, increase the **Height** property on your PointLight2D and DirectionalLight2D nodes. You may also want to increase the lights's **Energy** property slightly to get closer to how your lighting's intensity looked prior to enabling normal mapping.

Using additive sprites as a faster alternative to 2D lights

If you run into performance issues when using 2D lights, it may be worth replacing some of them with Sprite2D nodes that use additive blending. This is particularly suited for short-lived dynamic effects, such as bullets or explosions.

Additive sprites are much faster to render, since they don't need to go through a separate rendering pipeline. Additionally, it is possible to use this approach with AnimatedSprite2D (or Sprite2D + AnimationPlayer), which allows for animated 2D "lights" to be created.

However, additive sprites have a few downsides compared to 2D lights:

- The blending formula is inaccurate compared to "actual" 2D lighting. This is usually not a problem in sufficiently lit areas, but

this prevents additive sprites from correctly lighting up areas that are fully dark.
- Additive sprites cannot cast shadows, since they are not lights.
- Additive sprites ignore normal and specular maps used on other sprites.

To display a sprite with additive blending, create a Sprite2D node and assign a texture to it. In the inspector, scroll down to the **CanvasItem > Material** section, unfold it and click the dropdown next to the **Material** property. Choose **New CanvasItemMaterial**, click the newly created material to edit it, then set **Blend Mode** to **Add**.

2D meshes

Introduction

In 3D, meshes are used to display the world. In 2D, they are rare as images are used more often. Godot's 2D engine is a pure two-dimensional engine, so it can't really display 3D meshes directly (although it can be done via `Viewport` and `ViewportTexture`).

See also

If you are interested in displaying 3D meshes on a 2D viewport, see the Using a SubViewport as a texture tutorial.

2D meshes are meshes that contain two-dimensional geometry (Z can be omitted or ignored) instead of 3D. You can experiment creating them yourself using `SurfaceTool` from code and displaying them in a `MeshInstance2D` node.

Currently, the only way to generate a 2D mesh within the editor is by either importing an OBJ file as a mesh, or converting it from a Sprite2D.

Optimizing pixels drawn

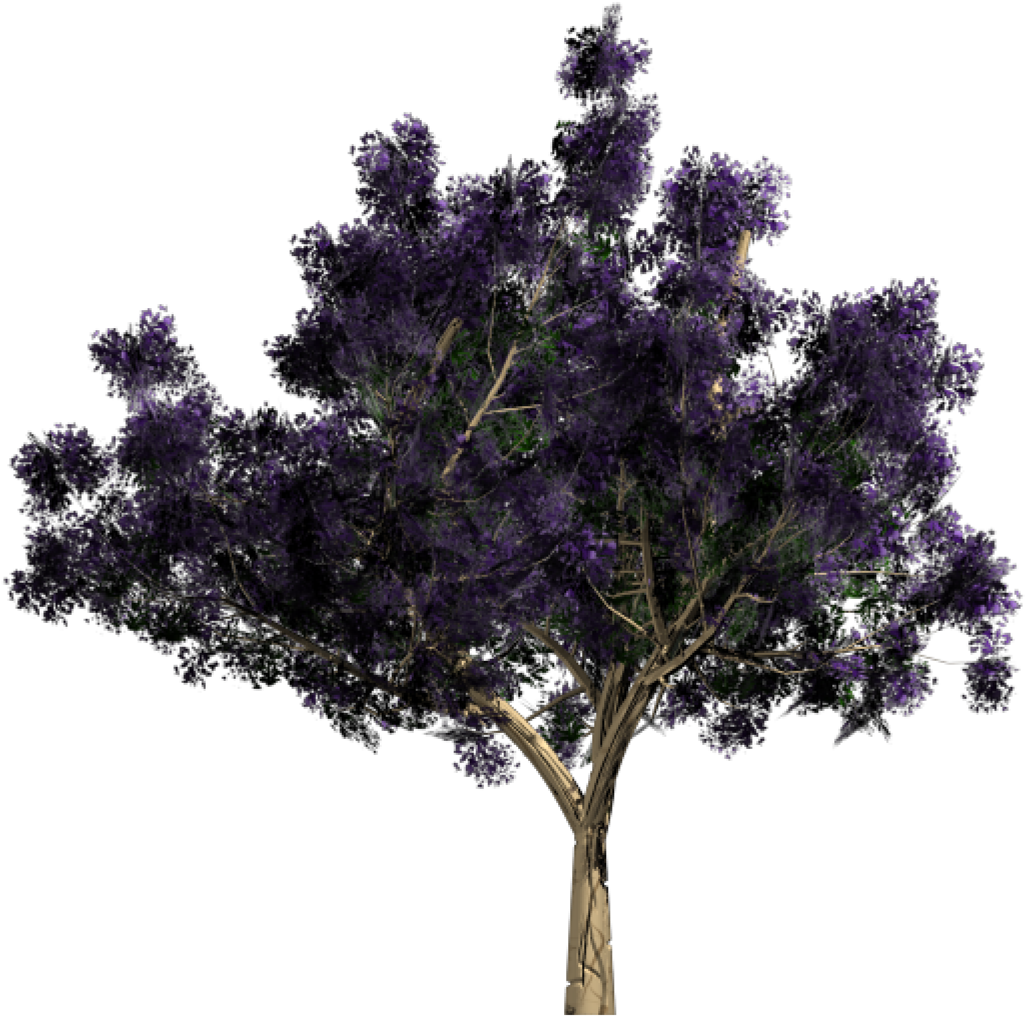
This workflow is useful for optimizing 2D drawing in some situations. When drawing large images with transparency, Godot will draw the whole quad to the screen. The large transparent areas will still be drawn.

This can affect performance, especially on mobile devices, when drawing very large images (generally screen sized), or layering multiple images on top of each other with large transparent areas (for example, when using `ParallaxBackground`).

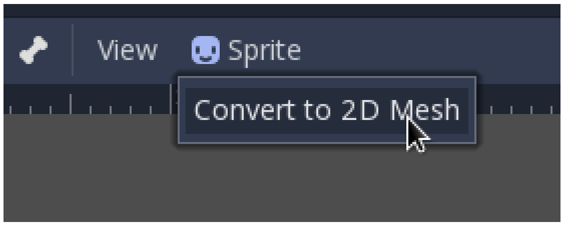
Converting to a mesh will ensure that only the opaque parts will be drawn and the rest will be ignored.

Converting Sprite2Ds to 2D meshes

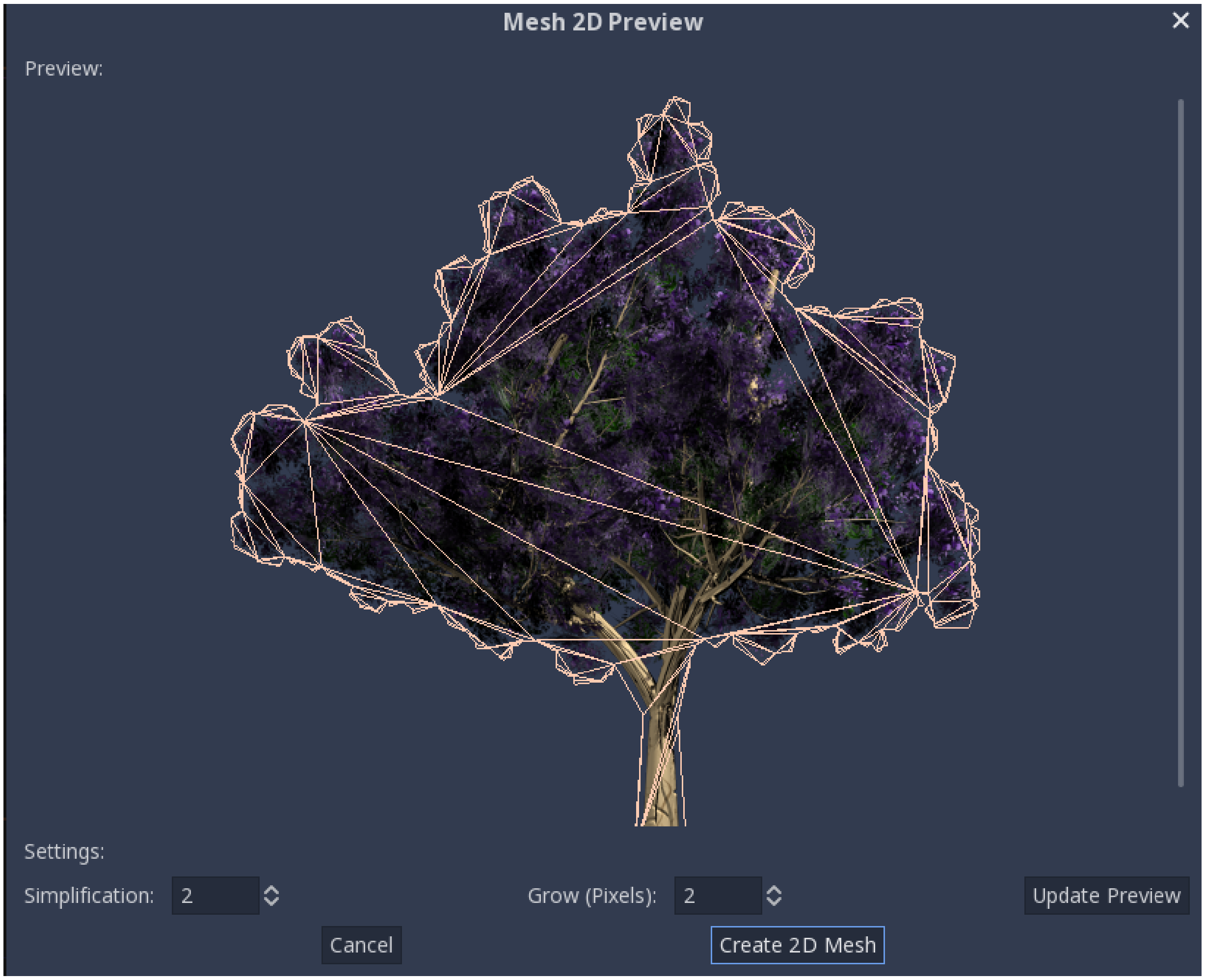
You can take advantage of this optimization by converting a `Sprite2D` to a `MeshInstance2D`. Start with an image that contains large amounts of transparency on the edges, like this tree:



Put it in a `Sprite2D` and select "Convert to 2D Mesh" from the menu:


View
Sprite
Convert to 2D Mesh

A dialog will appear, showing a preview of how the 2D mesh will be created:

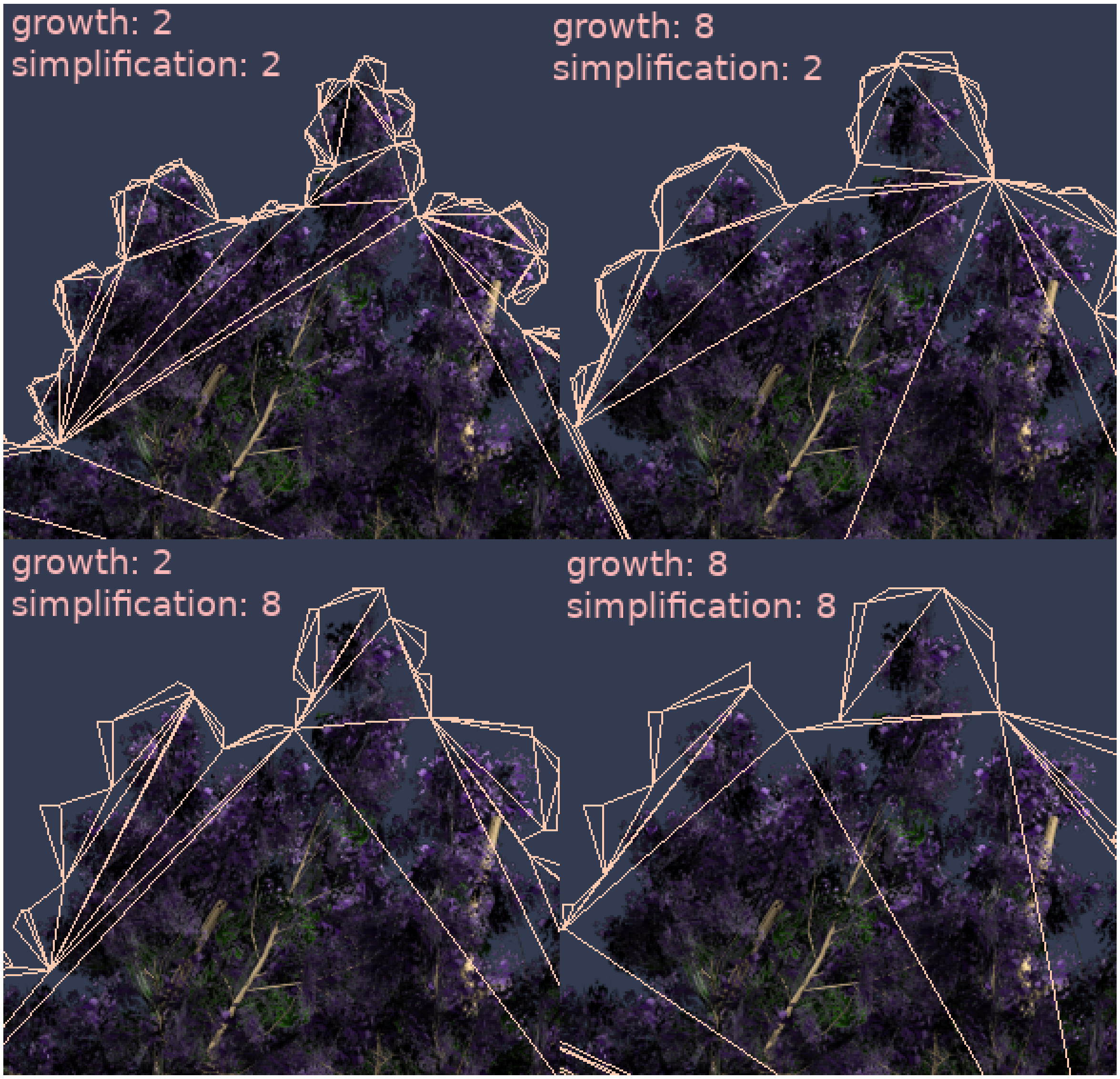


Mesh 2D Preview
Preview:
Settings:
Update Preview
Cancel
Create 2D Mesh

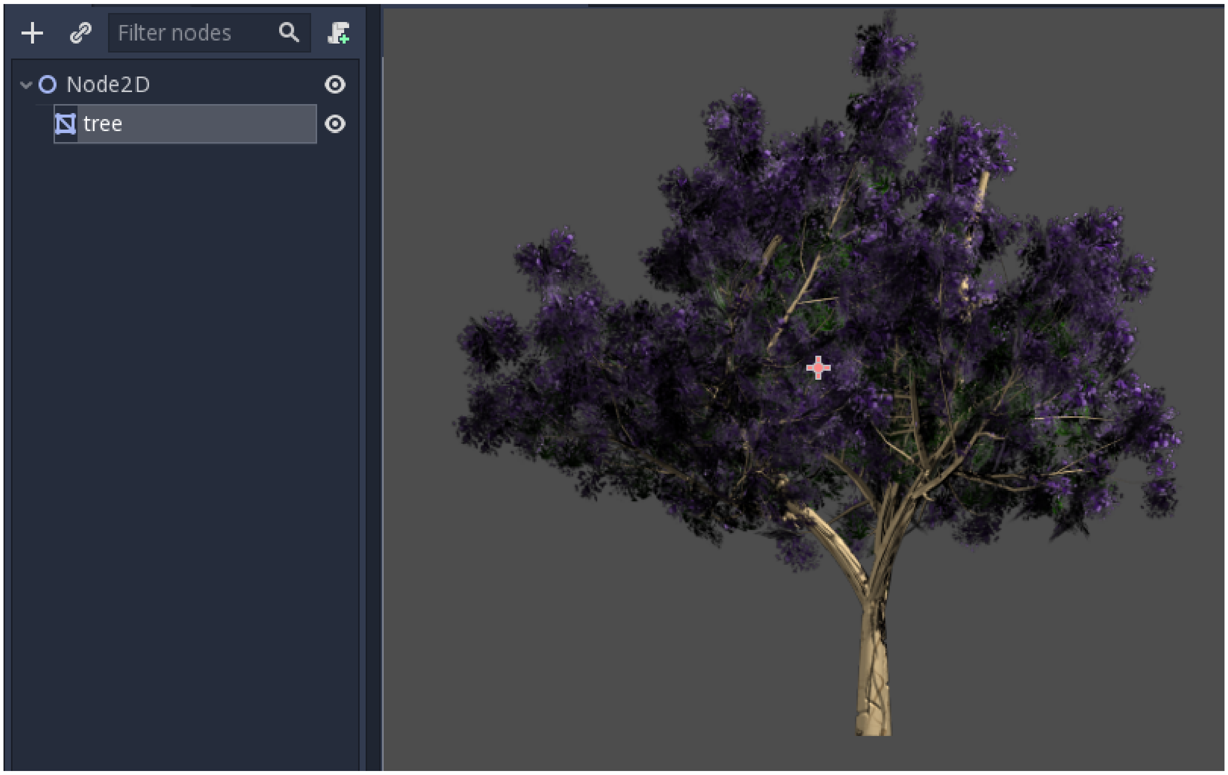
Simplification: 2

Grow (Pixels): 2

The default values are good enough for many cases, but you can change growth and simplification according to your needs:


growth: 2
simplification: 2
growth: 8
simplification: 2
growth: 2
simplification: 8
growth: 8
simplification: 8

Finally, push the `Convert 2D Mesh` button and your Sprite2D will be replaced:


Filter nodes
Node2D
tree

2D sprite animation

Introduction

In this tutorial, you'll learn how to create 2D animated characters with the AnimatedSprite2D class and the AnimationPlayer. Typically, when you create or download an animated character, it will come in one of two ways: as individual images or as a single sprite sheet containing all the animation's frames. Both can be animated in Godot with the AnimatedSprite2D class.

First, we'll use AnimatedSprite2D to animate a collection of individual images. Then we will animate a sprite sheet using this class. Finally, we will learn another way to animate a sprite sheet with AnimationPlayer and the *Animation* property of Sprite2D.

Note

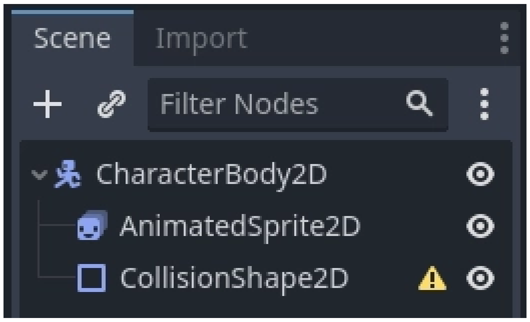
Art for the following examples by https://opengameart.org/users/ansimuz and tgfcoder.

Individual images with AnimatedSprite2D

In this scenario, you have a collection of images, each containing one of your character's animation frames. For this example, we'll use the following animation:

You can download the images here: [2d_sprite_animation_assets.zip](https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/2d_sprite_animation_assets.zip) [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/2d_sprite_animation_assets.zip]

Unzip the images and place them in your project folder. Set up your scene tree with the following nodes:

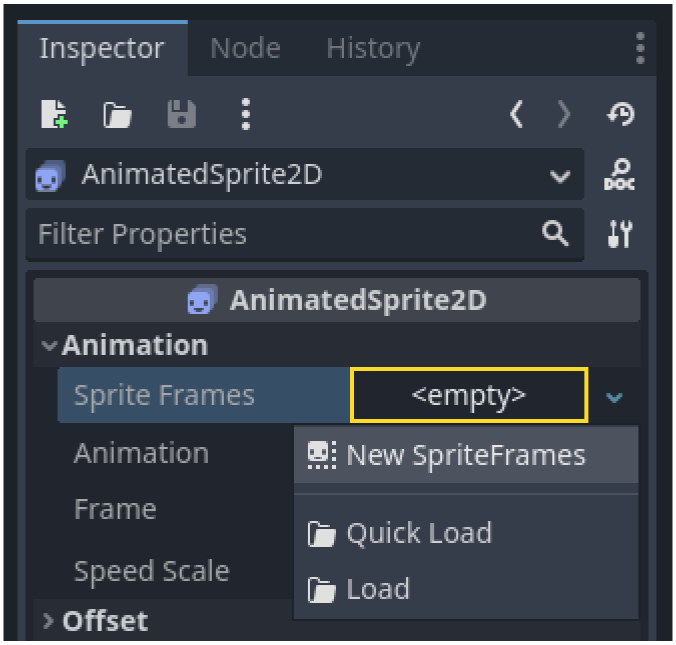


CharacterBody2D
AnimatedSprite2D
CollisionShape2D

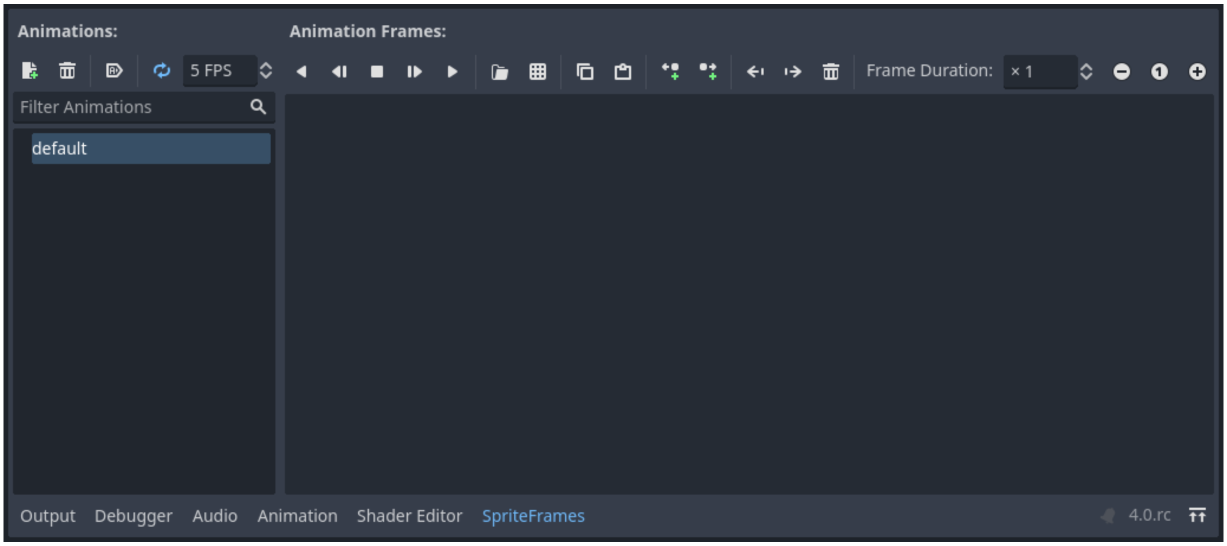
Note

The root node could also be [Area2D]() or [RigidBody2D](). The animation will still be made in the same way. Once the animation is completed, you can assign a shape to the CollisionShape2D. See [Physics Introduction]() for more information.

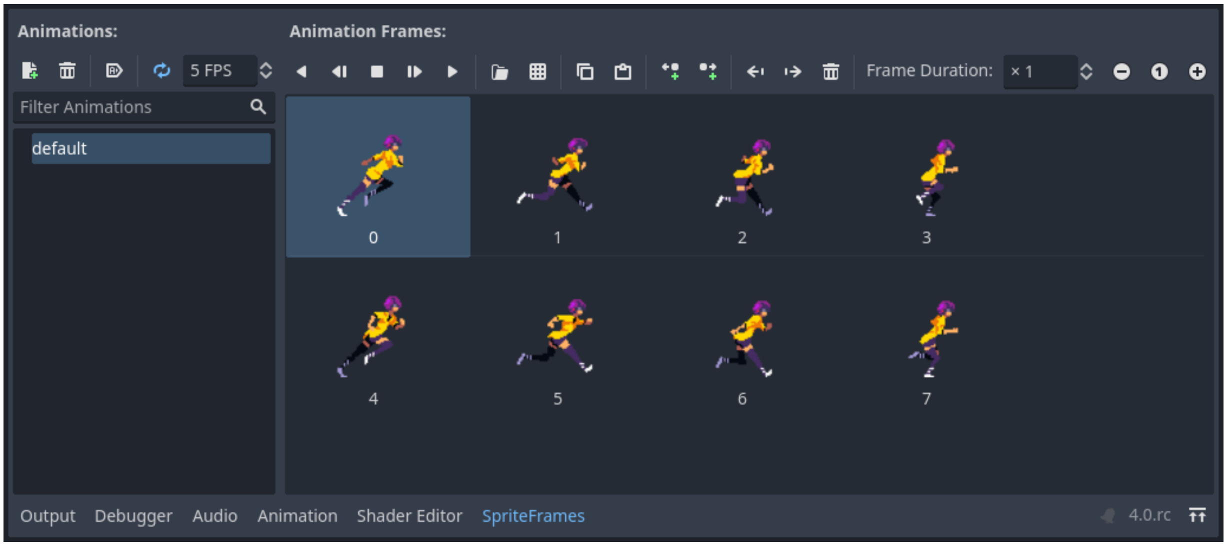
Now select the `AnimatedSprite2D` and in its *SpriteFrames* property, select "New SpriteFrames".


Inspector
Node
History
AnimatedSprite2D
Filter Properties
AnimatedSprite2D
Animation
Sprite Frames
<empty>
Animation
Frame
Speed Scale
Offset
New SpriteFrames
Quick Load
Load

Click on the new SpriteFrames resource and you'll see a new panel appear at the bottom of the editor window:


Animations:
Animation Frames:
5 FPS
Filter Animations
default
Frame Duration:
× 1
Output
Debugger
Audio
Animation
Shader Editor
SpriteFrames
4.0.rc

From the FileSystem dock on the left side, drag the 8 individual images into the center part of the SpriteFrames panel. On the left side, change the name of the animation from "default" to "run".


Animations:
Animation Frames:
5 FPS
Frame Duration:
× 1
Filter Animations
default
0
1
2
3
4
5
6
7
Output
Debugger
Audio
Animation
Shader Editor
SpriteFrames
4.0.rc

Use the "Play" buttons on the top-right of the *Filter Animations* input to preview the animation. You should now see the animation playing in the viewport. However, it is a bit slow. To fix this, change the *Speed (FPS)* setting in the SpriteFrames panel to 10.

You can add additional animations by clicking the "Add Animation" button and adding additional images.

Controlling the animation

Once the animation is complete, you can control the animation via code using the `play()` and `stop()` methods. Here is a brief example to play the animation while the right arrow key is held, and stop it when the key is released.

GDScript

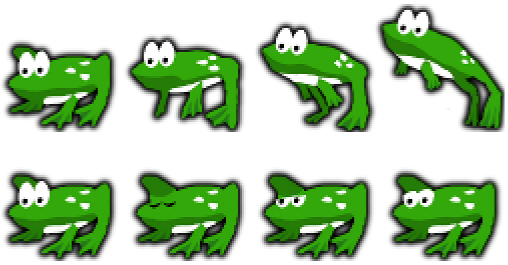
```
extends CharacterBody2D

@onready var _animated_sprite = $AnimatedSprite2D

func _process(_delta):
    if Input.is_action_pressed("ui_right"):
        _animated_sprite.play("run")
    else:
        _animated_sprite.stop()
```

C#

```
using Godot;

public partial class Character : CharacterBody2D
{
    private AnimatedSprite2D _animatedSprite;

    public override void _Ready()
    {
        _animatedSprite = GetNode<AnimatedSprite2D>
("AnimatedSprite2D");
    }

    public override _Process(float _delta)
    {
        if (Input.IsActionPressed("ui_right"))
        {
            _animatedSprite.Play("run");
        }
        else
        {
            _animatedSprite.Stop();
        }
    }
}
```

Sprite sheet with AnimatedSprite2D

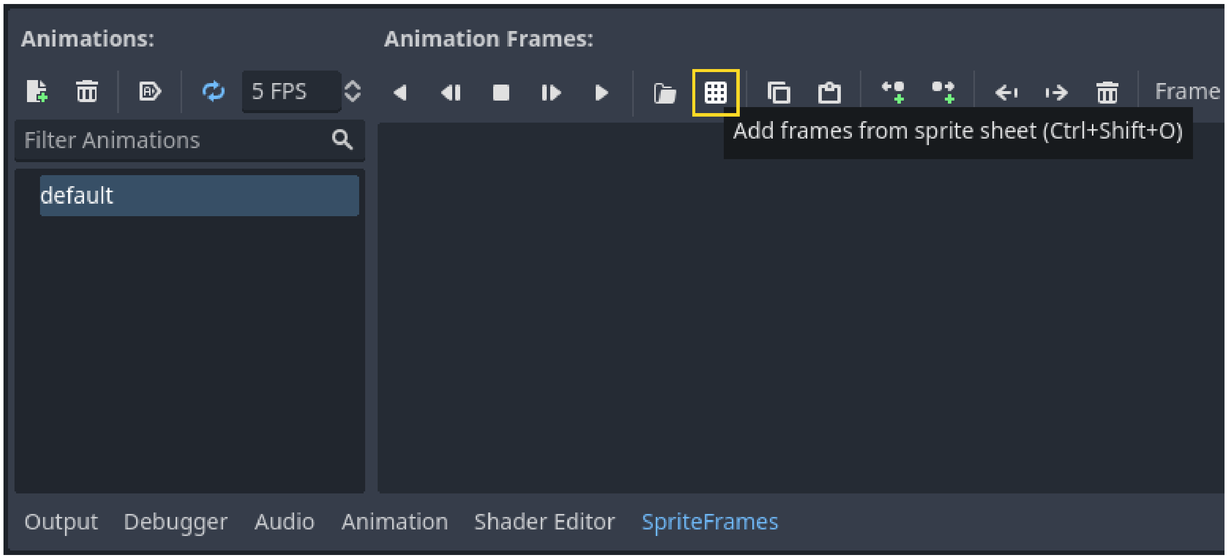
You can also easily animate from a sprite sheet with the class `AnimatedSprite2D`. We will use this public domain sprite sheet:



Right-click the image and choose "Save Image As" to download it, and then copy the image into your project folder.

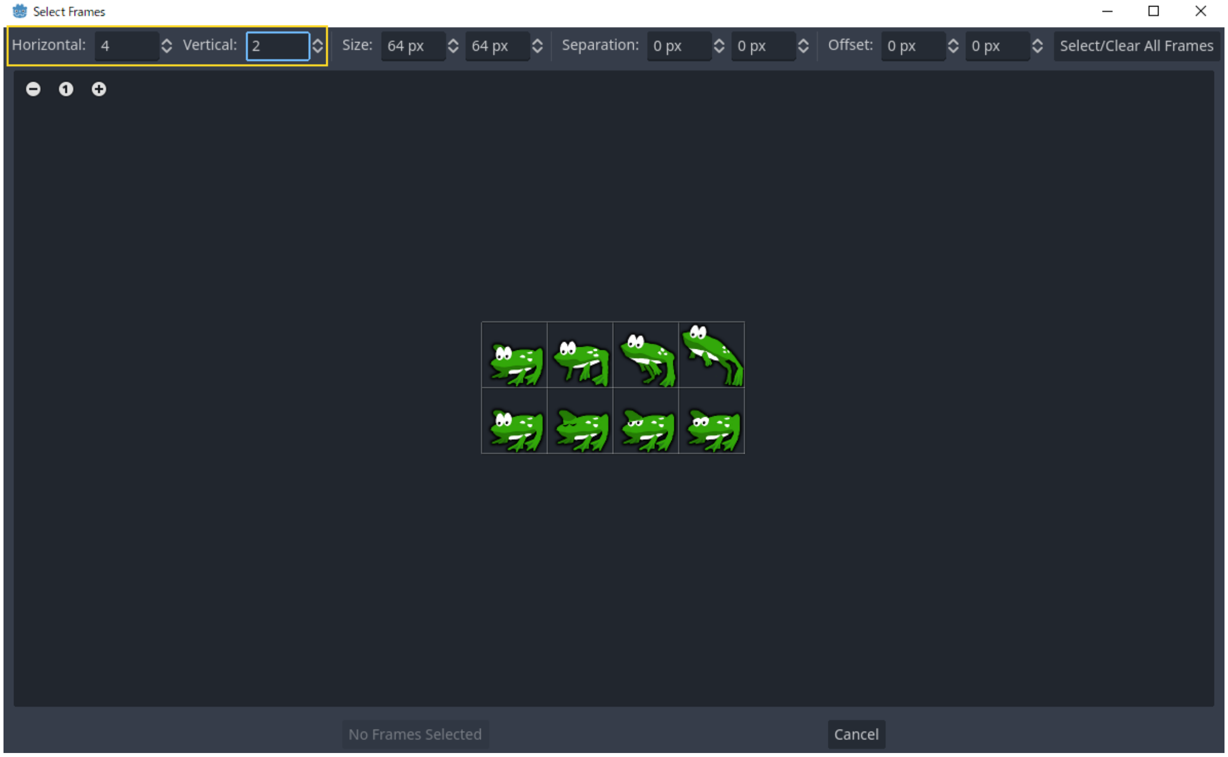
Set up your scene tree the same way you did previously when using individual images. Select the `AnimatedSprite2D` and in its *SpriteFrames* property, select "New SpriteFrames".

Click on the new SpriteFrames resource. This time, when the bottom panel appears, select "Add frames from a Sprite Sheet".

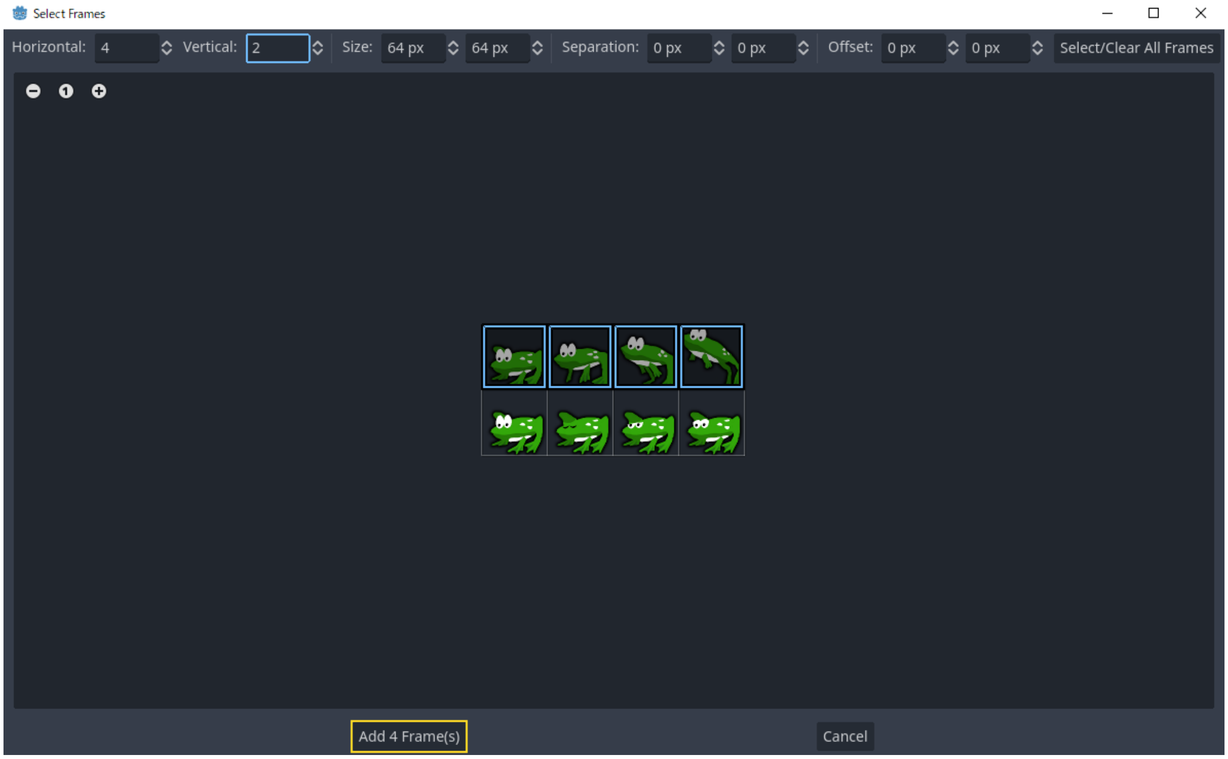

Animations:
Animation Frames:
5 FPS
Frame
Filter Animations
default
Add frames from sprite sheet (Ctrl+Shift+O)
Output
Debugger
Audio
Animation
Shader Editor
SpriteFrames

You will be prompted to open a file. Select your sprite sheet.

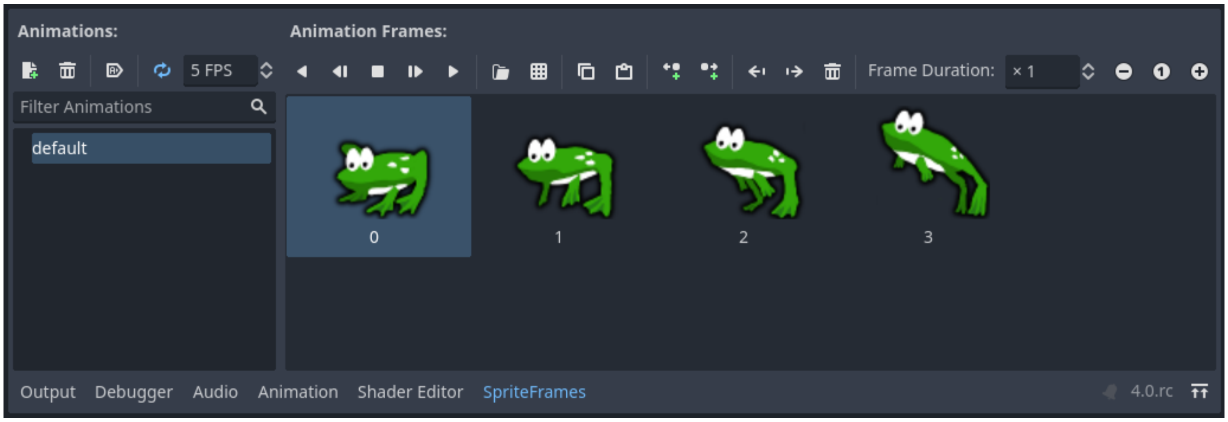
A new window will open, showing your sprite sheet. The first thing you will need to do is to change the number of vertical and horizontal images in your sprite sheet. In this sprite sheet, we have four images horizontally and two images vertically.


Select Frames
Horizontal: 4
Vertical: 2
Size: 64 px 64 px
Separation: 0 px 0 px
Offset: 0 px 0 px
Select/Clear All Frames
No Frames Selected
Cancel

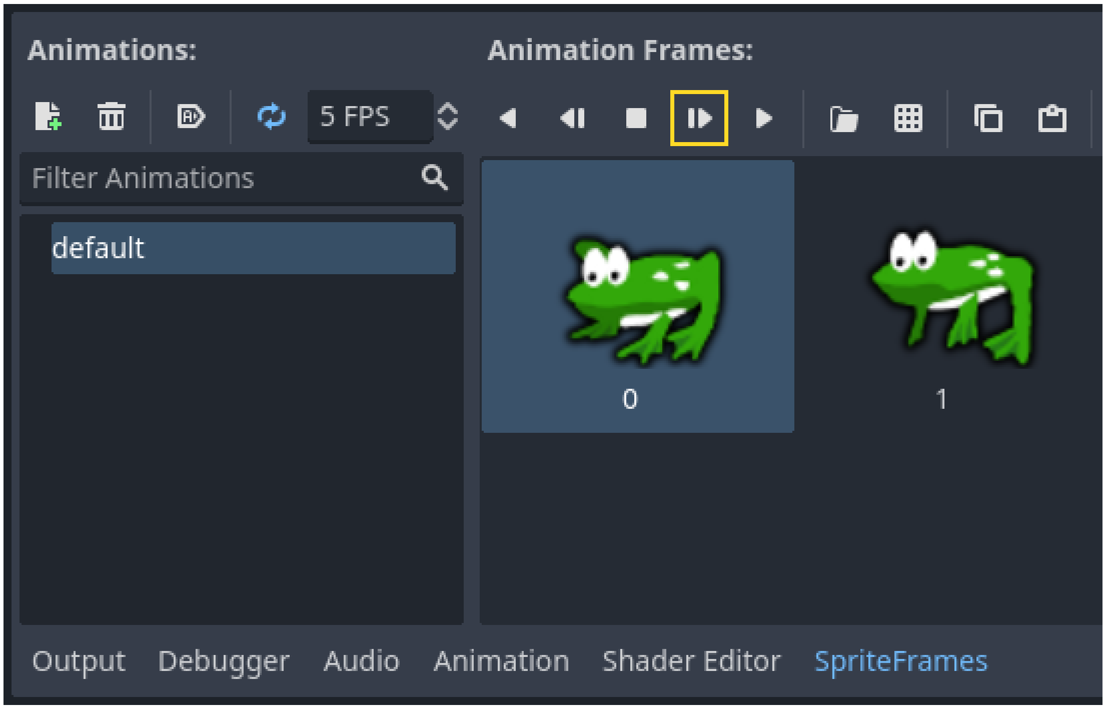
Next, select the frames from the sprite sheet that you want to include in your animation. We will select the top four, then click "Add 4 frames" to create the animation.


Select Frames
Horizontal: 4
Vertical: 2
Size: 64 px 64 px
Separation: 0 px 0 px
Offset: 0 px 0 px
Select/Clear All Frames
Add 4 Frame(s)
Cancel

You will now see your animation under the list of animations in the bottom panel. Double click on default to change the name of the animation to jump.


Animations:
Animation Frames:
5 FPS
Frame Duration: × 1
Filter Animations
default
0
1
2
3
Output
Debugger
Audio
Animation
Shader Editor
SpriteFrames
4.0.rc

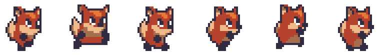
Finally, check the play button on the SpriteFrames editor to see your frog jump!


Animations:
Animation Frames:
5 FPS
Filter Animations
default
0
1
Output
Debugger
Audio
Animation
Shader Editor
SpriteFrames

Sprite sheet with AnimationPlayer

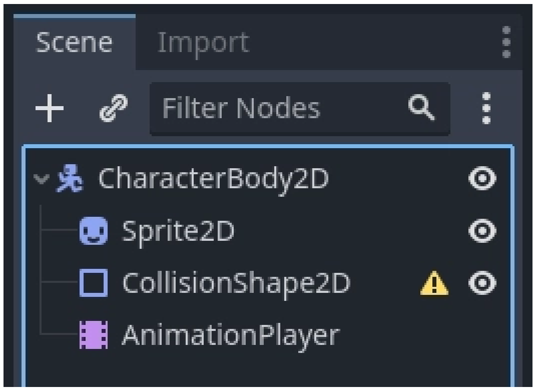
Another way that you can animate when using a sprite sheet is to use a standard Sprite2D node to display the texture, and then animating the change from texture to texture with AnimationPlayer.

Consider this sprite sheet, which contains 6 frames of animation:



Right-click the image and choose "Save Image As" to download, then copy the image into your project folder.

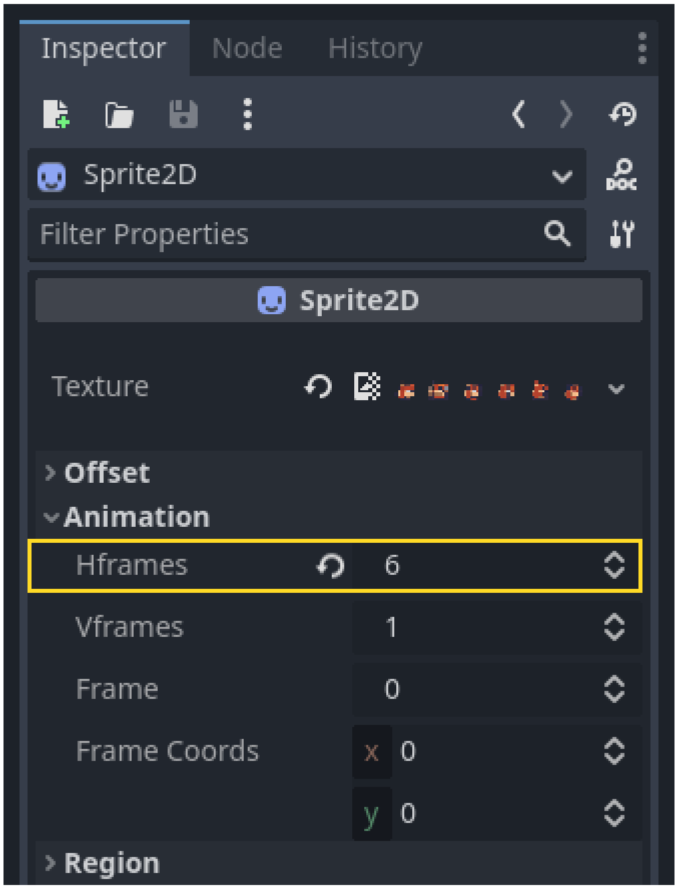
Our goal is to display these images one after another in a loop. Start by setting up your scene tree:


Scene
Import
Filter Nodes
CharacterBody2D
Sprite2D
CollisionShape2D
AnimationPlayer

Note

The root node could also be Area2D or RigidBody2D. The animation will still be made in the same way. Once the animation is completed, you can assign a shape to the CollisionShape2D. See Physics Introduction for more information.

Drag the spritesheet into the Sprite's *Texture* property, and you'll see the whole sheet displayed on the screen. To slice it up into individual frames, expand the *Animation* section in the Inspector and set the *Hframes* to `6`. *Hframes* and *Vframes* are the number of horizontal and vertical frames in your sprite sheet.


Inspector
Node
History
Sprite2D
Filter Properties
Sprite2D
Texture
Offset
Animation
Region

Hframes 6

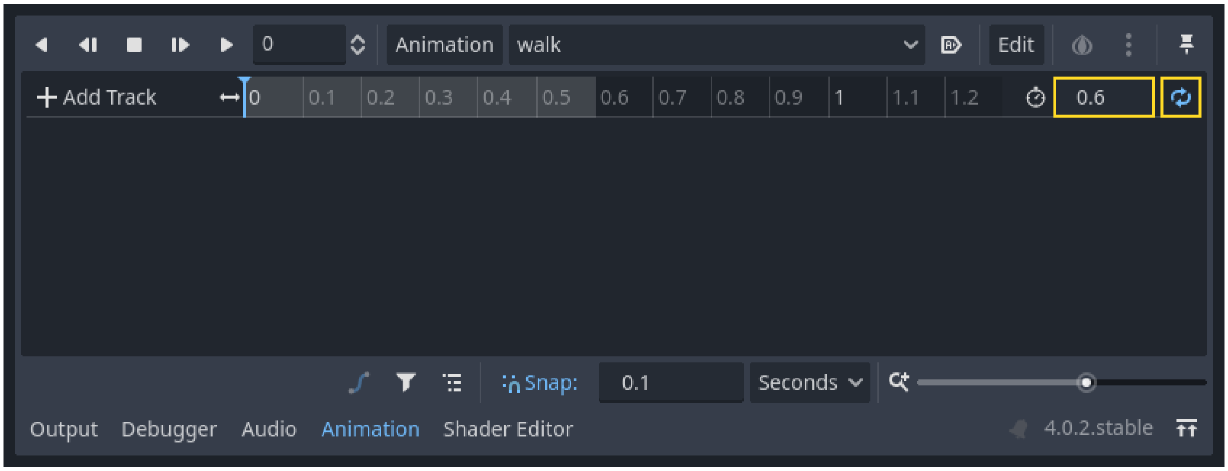
Vframes 1

Frame 0

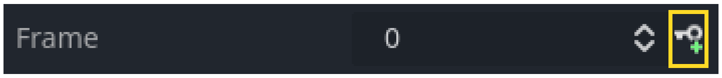
Frame Coords x 0 y 0

Now try changing the value of the *Frame* property. You'll see that it ranges from `0` to `5` and the image displayed by the Sprite2D changes accordingly. This is the property we'll be animating.

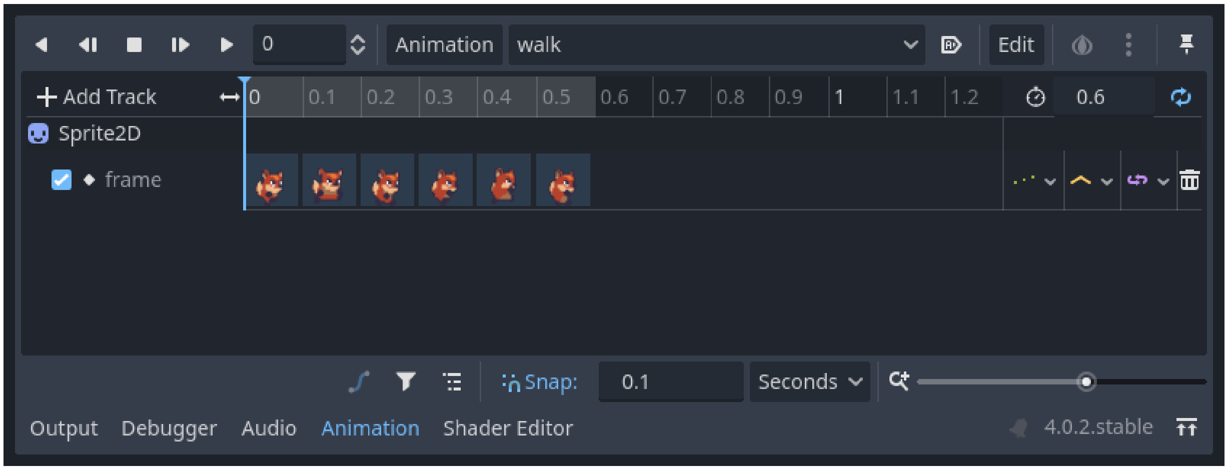
Select the `AnimationPlayer` and click the "Animation" button followed by "New". Name the new animation "walk". Set the animation length to `0.6` and click the "Loop" button so that our animation will repeat.


0
Animation
walk
Edit
+ Add Track
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
1.2
0.6
Snap:
0.1
Seconds
Output
Debugger
Audio
Animation
Shader Editor
4.0.2.stable

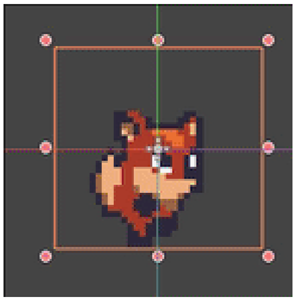
Now select the `Sprite2D` node and click the key icon to add a new track.



Continue adding frames at each point in the timeline (`0.1` seconds by default), until you have all the frames from 0 to 5. You'll see the frames actually appearing in the animation track:


0
Animation
walk
Edit
+ Add Track
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
1.2
0.6
Sprite2D
frame
Snap:
0.1
Seconds
Output
Debugger
Audio
Animation
Shader Editor
4.0.2.stable

Press "Play" on the animation to see how it looks.



Controlling an AnimationPlayer animation

Like with AnimatedSprite2D, you can control the animation via code using the `play()` and `stop()` methods. Again, here is an example to play the animation while the right arrow key is held, and stop it when the key is released.

GDScript

```
extends CharacterBody2D

@onready var _animation_player = $AnimationPlayer

func _process(_delta):
    if Input.is_action_pressed("ui_right"):
        _animation_player.play("walk")
    else:
        _animation_player.stop()
```

C#

```
using Godot;

public partial class Character : CharacterBody2D
{
    private AnimationPlayer _animationPlayer;

    public override void _Ready()
    {
        _animationPlayer = GetNode<AnimationPlayer>
("AnimationPlayer");
    }
    public override void _Process(float _delta)
    {
        if (Input.IsActionPressed("ui_right"))
        {
            _animationPlayer.Play("walk");
        }
        else
        {
            _animationPlayer.Stop();
        }
    }
}
```

Note

If updating both an animation and a separate property at once (for example, a platformer may update the sprite's `h_flip`/`v_flip` properties when a character turns while starting a 'turning' animation), it's important to keep in mind that `play()` isn't applied instantly. Instead, it's applied the next time the AnimationPlayer is processed. This may end up being on the next frame, causing a 'glitch' frame, where the property change was applied but the animation was not. If this turns out to be a problem, after calling `play()`, you can call `advance(0)` to update the animation immediately.

Summary

These examples illustrate the two classes you can use in Godot for 2D animation. `AnimationPlayer` is a bit more complex than `AnimatedSprite2D`, but it provides additional functionality, since you can also animate other properties like position or scale. The class `AnimationPlayer` can also be used with an `AnimatedSprite2D`. Experiment to see what works best for your needs.

2D particle systems

Intro

Particle systems are used to simulate complex physical effects, such as sparks, fire, magic particles, smoke, mist, etc.

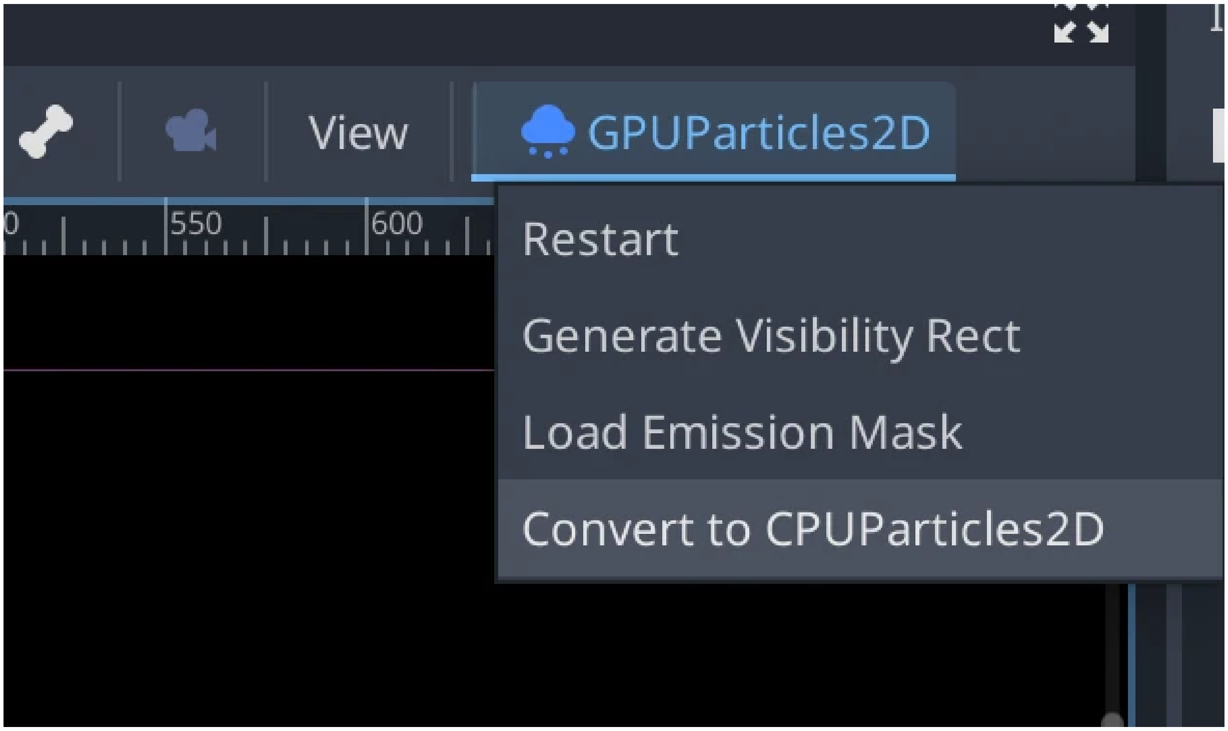
The idea is that a "particle" is emitted at a fixed interval and with a fixed lifetime. During its lifetime, every particle will have the same base behavior. What makes each particle different from the rest and provides a more organic look is the "randomness" associated with each parameter. In essence, creating a particle system means setting base physics parameters and then adding randomness to them.

Particle nodes

Godot provides two different nodes for 2D particles, GPUParticles2D and CPUParticles2D. GPUParticles2D is more advanced and uses the GPU to process particle effects. CPUParticles2D is a CPU-driven option with near-feature parity with GPUParticles2D, but lower performance when using large amounts of particles. On the other hand, CPUParticles2D may perform better on low-end systems or in GPU-bottlenecked situations.

While GPUParticles2D is configured via a ParticleProcessMaterial (and optionally with a custom shader), the matching options are provided via node properties in CPUParticles2D (with the exception of the trail settings).

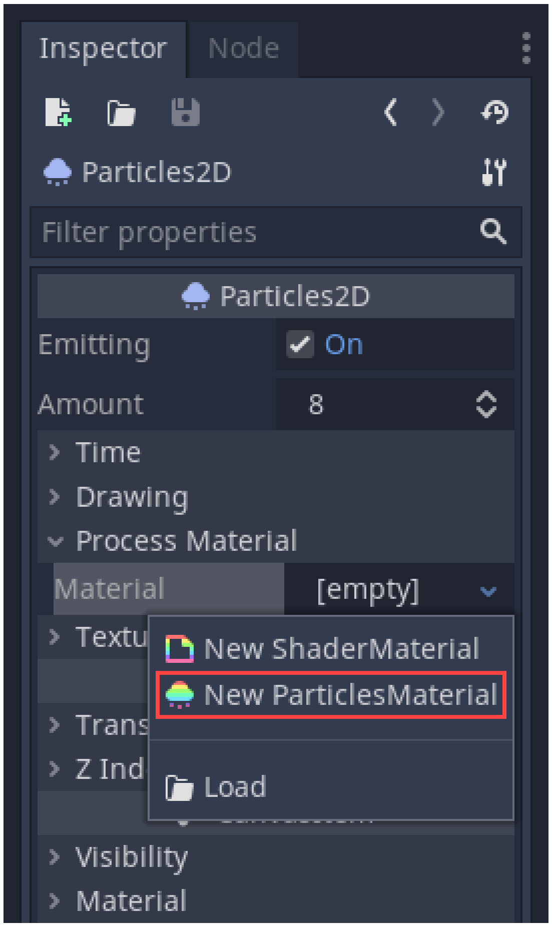
You can convert a GPUParticles2D node into a CPUParticles2D node by clicking on the node in the inspector, selecting the 2D viewport, and selecting **GPUParticles2D > Convert to CPUParticles2D** in the viewport toolbar.


View
GPUParticles2D
550
600
Restart
Generate Visibility Rect
Load Emission Mask
Convert to CPUParticles2D

The rest of this tutorial is going to use the GPUParticles2D node. First, add a GPUParticles2D node to your scene. After creating that node you will notice that only a white dot was created, and that there is a warning icon next to your GPUParticles2D node in the scene dock. This is because the node needs a ParticleProcessMaterial to function.

ParticleProcessMaterial

To add a process material to your particles node, go to `Process Material` in your inspector panel. Click on the box next to `Material`, and from the dropdown menu select `New ParticleProcessMaterial`.

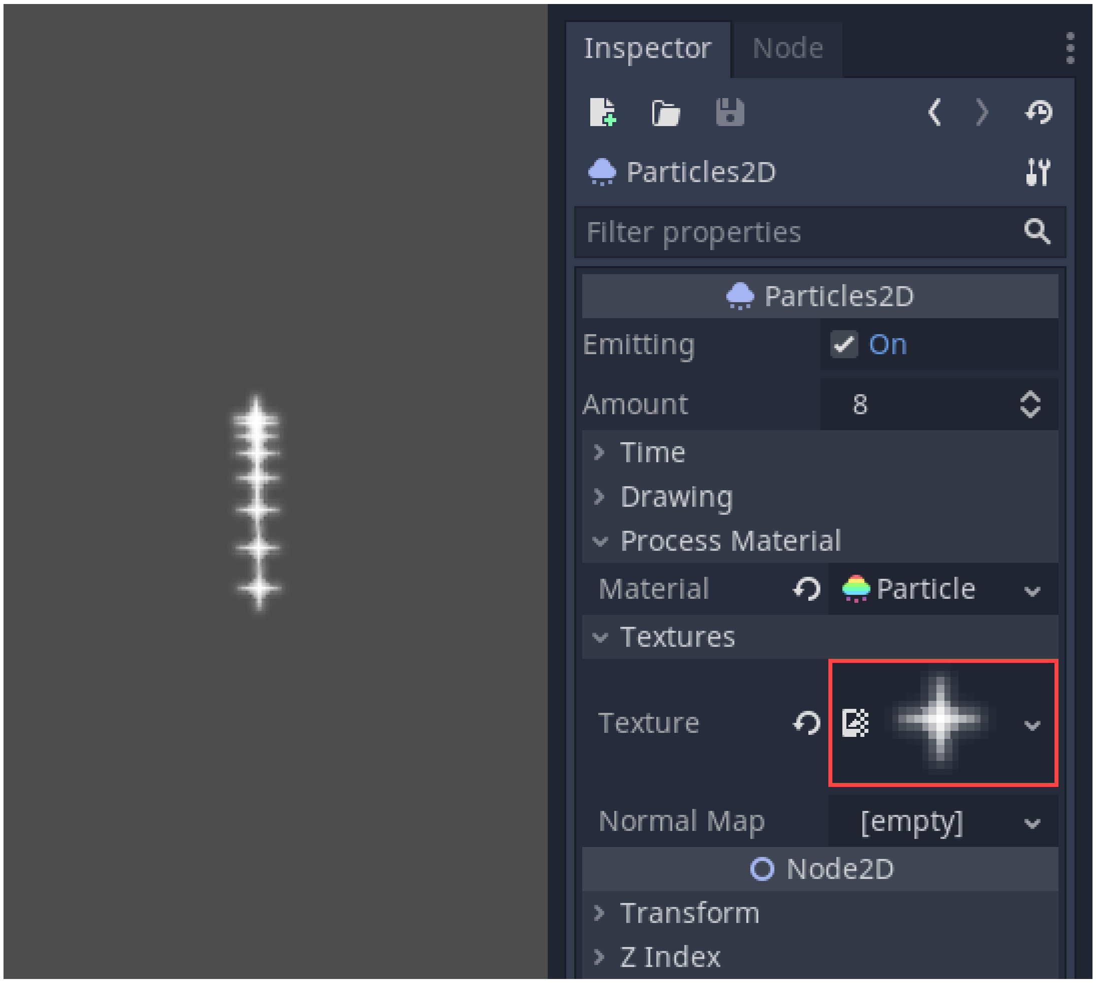

Inspector
Node
Particles2D
Filter properties
Particles2D
Emitting
On
Amount
8
Time
Drawing
Process Material
Material
[empty]
Textu
Trans
Z Ind
Visibility
Material
New ShaderMaterial
New ParticlesMaterial
Load

Your GPUParticles2D node should now be emitting white points downward.

Texture

A particle system can use a single texture or an animation *flipbook*. A flipbook is a texture that contains several frames of animation that can be played back, or chosen at random during emission. This is equivalent to a spritesheet for particles.

The texture is set via the **Texture** property:


Inspector
Node
Particles2D
Filter properties
Particles2D
Time
Drawing
Process Material
Textures
Texture
Node2D
Transform
Z Index

Emitting ☑ On

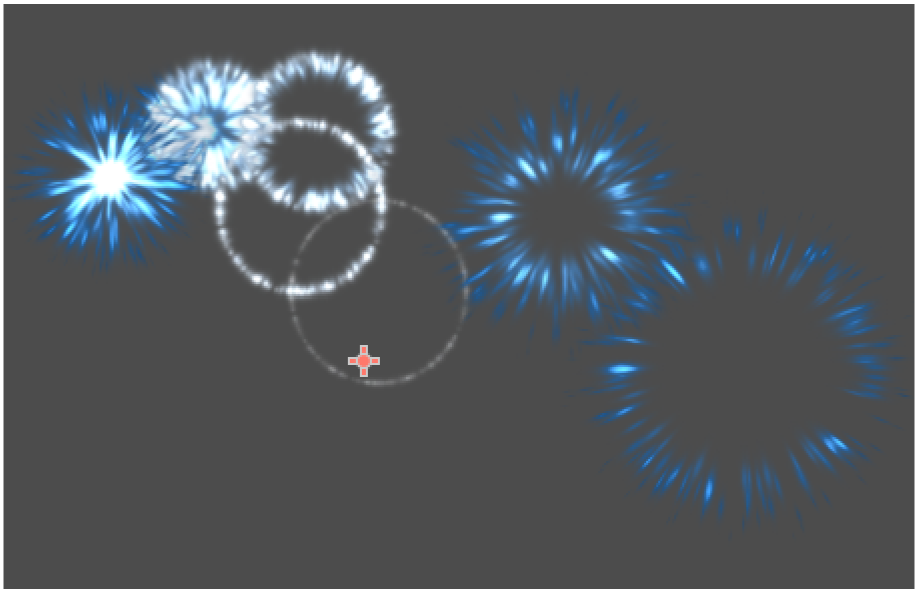
Amount 8

Material Particle

Normal Map [empty]

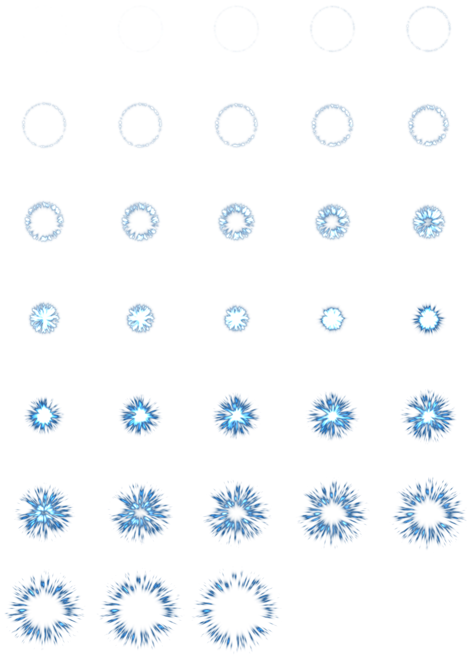
Using an animation flipbook

Particle flipbooks are suited to reproduce complex effects such as smoke, fire, explosions. They can also be used to introduce random texture variation, by making every particle use a different texture. You can find existing particle flipbook images online, or pre-render them using external tools such as Blender [https://www.blender.org/] or EmberGen [https://jangafx.com/software/embergen/].



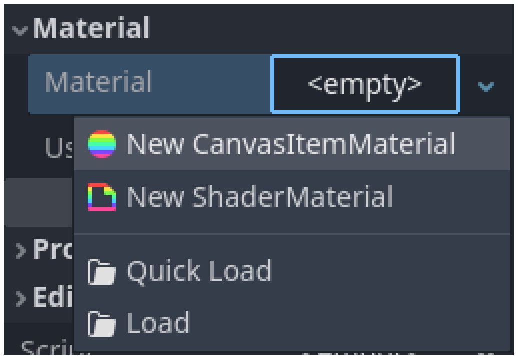
Example of a particle system that uses a flipbook texture

Using an animation flipbook requires additional configuration compared to a single texture. For demonstration purposes, we'll use this texture with 5 columns and 7 rows (right-click and choose **Save as...**):



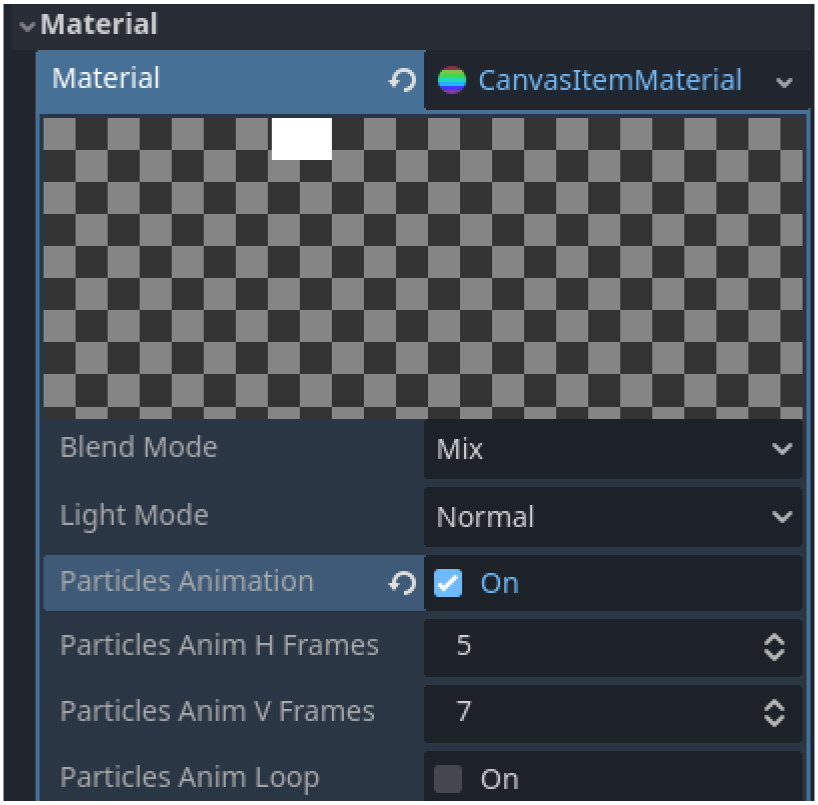
Credit: JoesAlotofthings [https://opengameart.org/content/alot-of-particles-indispersal-special-effect-alotofparticles30] (CC BY 4.0)

To use an animation flipbook, you must create a new CanvasItemMaterial in the Material section of the GPUParticles2D (or CPUParticles2D) node:


Material
Material
<empty>
Us
New CanvasItemMaterial
New ShaderMaterial
Quick Load
Load
Pro
Edi
Scri

Creating a CanvasItemMaterial at the bottom of the particles node inspector

In this CanvasItemMaterial, enable **Particle Animation** and set **H Frames** and **V Frames** to the number of columns and rows present in your flipbook texture:



Material

Material: CanvasItemMaterial

Blend Mode: Mix

Light Mode: Normal

Particles Animation: ☑ On

Particles Anim H Frames: 5

Particles Anim V Frames: 7

Particles Anim Loop: ☐ On

Configuring the CanvasItemMaterial for the example flipbook texture

Once this is done, the Animation section in ParticleProcessMaterial (for GPUParticles2D) or in the CPUParticles2D inspector will be effective.

Tip

If your flipbook texture has a black background instead of a transparent background, you will also need to set the blend mode to **Add** instead of **Mix** for correct display. Alternatively, you can modify the texture to have a transparent background in an image editor. In GIMP [https://gimp.org], this can be done using the **Color > Color to Alpha** menu.

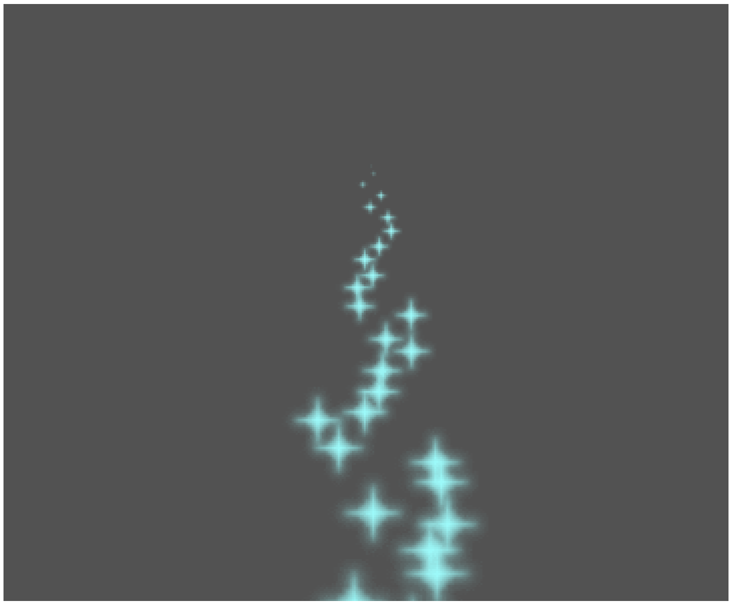
Time parameters

Lifetime

The time in seconds that every particle will stay alive. When lifetime ends, a new particle is created to replace it.

Lifetime: 0.5

Lifetime: 4.0



One Shot

When enabled, a GPUParticles2D node will emit all of its particles once and then never again.

Preprocess

Particle systems begin with zero particles emitted, then start emitting. This can be an inconvenience when loading a scene and systems like a torch, mist, etc. begin emitting the moment you enter. Preprocess is used to let the system process a given number of seconds before it is actually drawn the first time.

Speed Scale

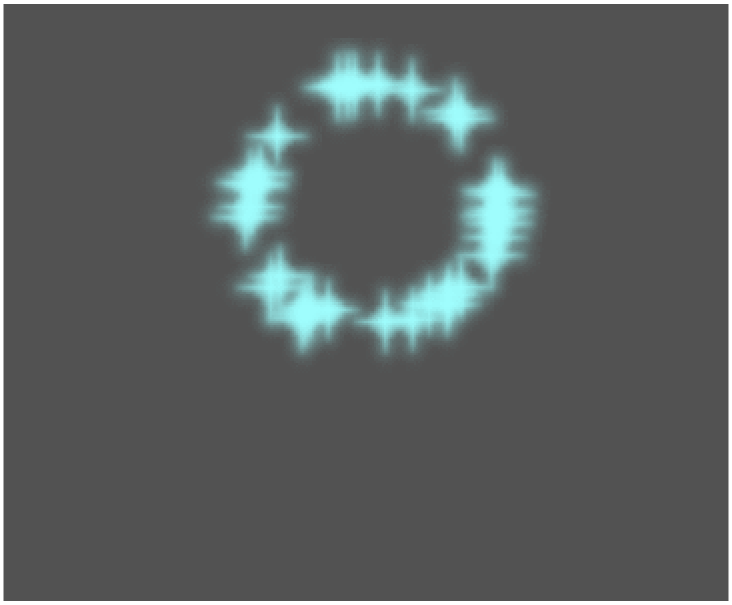
The speed scale has a default value of `1` and is used to adjust the speed of a particle system. Lowering the value will make the particles slower while increasing the value will make the particles much faster.

Explosiveness

If lifetime is `1` and there are 10 particles, it means a particle will be emitted every 0.1 seconds. The explosiveness parameter changes this, and forces particles to be emitted all together. Ranges are:

- 0: Emit particles at regular intervals (default value).
- 1: Emit all particles simultaneously.

Values in the middle are also allowed. This feature is useful for creating explosions or sudden bursts of particles:



Randomness

All physics parameters can be randomized. Random values range from `0` to `1`. The formula to randomize a parameter is:

```
initial_value = param_value + param_value * randomness
```

Fixed FPS

This setting can be used to set the particle system to render at a fixed FPS. For instance, changing the value to `2` will make the particles render at 2 frames per second. Note this does not slow down the particle system itself.

Note

Godot 4.3 does not currently support physics interpolation for 2D particles. As a workaround, disable physics interpolation for the particles node by setting **Node > Physics Interpolation > Mode** at the bottom of the inspector.

Fract Delta

This can be used to turn Fract Delta on or off.

Drawing parameters

Visibility Rect

The visibility rectangle controls the visibility of the particles on screen. If this rectangle is outside of the viewport, the engine will not render the particles on screen.

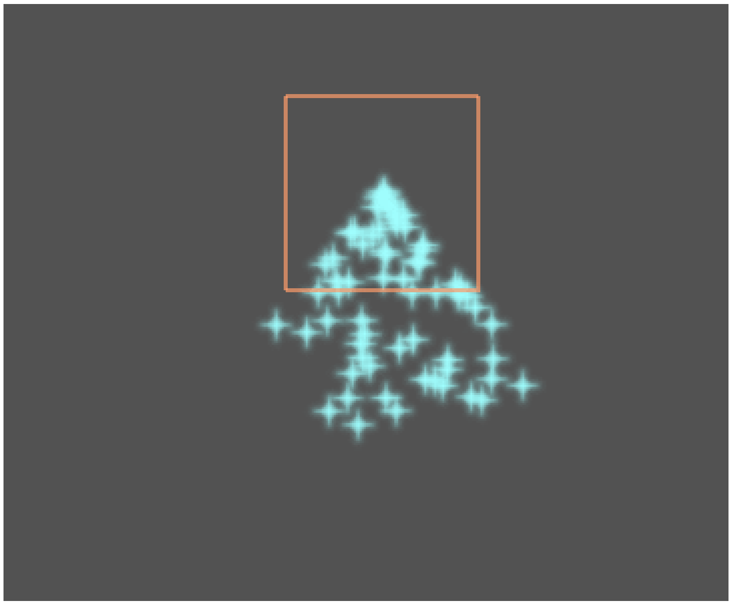
The rectangle's `W` and `H` properties respectively control its Width and its Height. The `X` and `Y` properties control the position of the upper-left corner of the rectangle, relative to the particle emitter.

You can have Godot generate a Visibility Rect automatically using the toolbar above the 2d view. To do so, select the GPUParticles2D node and Click `Particles > Generate Visibility Rect`. Godot will simulate the Particles2D node emitting particles for a few seconds and set the rectangle to fit the surface the particles take.

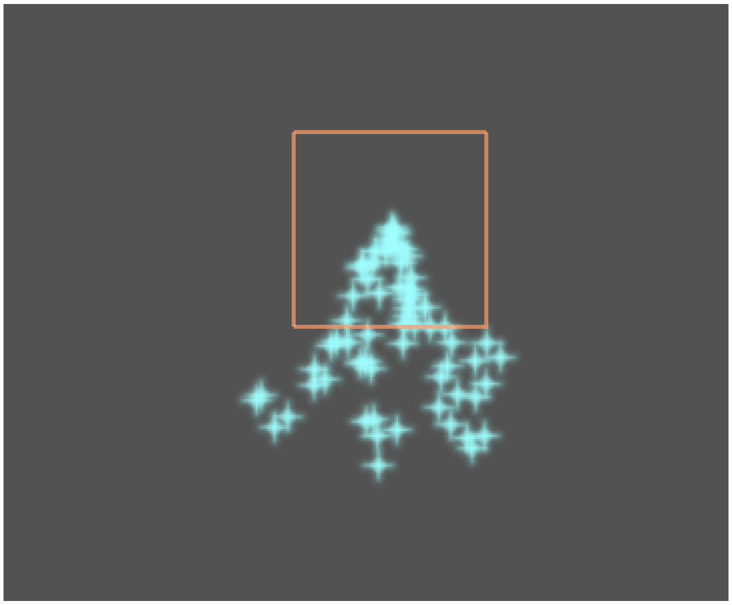
You can control the emit duration with the `Generation Time (sec)` option. The maximum value is 25 seconds. If you need more time for your particles to move around, you can temporarily change the `preprocess` duration on the Particles2D node.

Local Coords

By default this option is on, and it means that the space that particles are emitted to is relative to the node. If the node is moved, all particles are moved with it:



If disabled, particles will emit to global space, meaning that if the node is moved, already emitted particles are not affected:



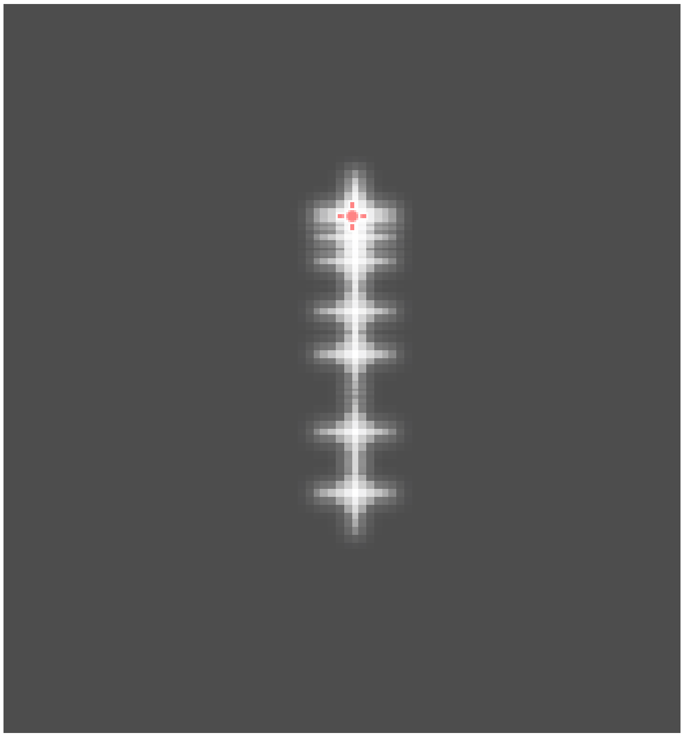
Draw Order

This controls the order in which individual particles are drawn. `Index` means particles are drawn according to their emission order (default). `Lifetime` means they are drawn in order of remaining lifetime.

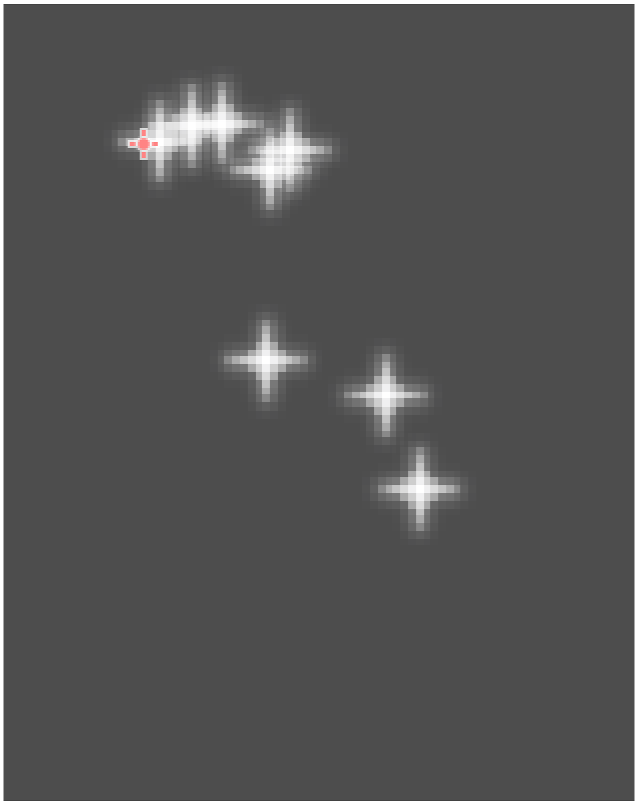
ParticleProcessMaterial settings

Direction

This is the base direction at which particles emit. The default is `Vector3(1, 0, 0)` which makes particles emit to the right. However, with the default gravity settings, particles will go straight down.



For this property to be noticeable, you need an *initial velocity* greater than 0. Here, we set the initial velocity to 40. You'll notice that particles emit toward the right, then go down because of gravity.



Spread

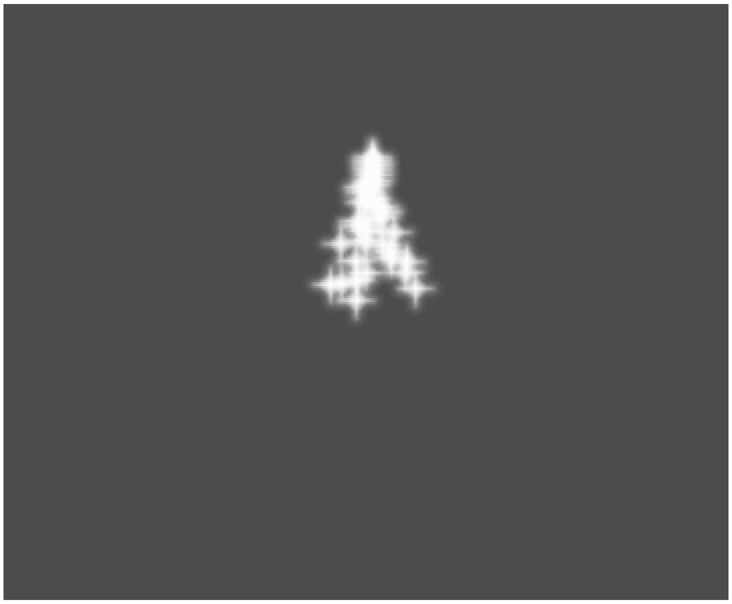
This parameter is the angle in degrees which will be randomly added in either direction to the base `Direction`. A spread of `180` will emit in all directions (+/- 180). For spread to do anything the "Initial Velocity" parameter must be greater than 0.

Flatness

This property is only useful for 3D particles.

Gravity

The gravity applied to every particle.



Initial Velocity

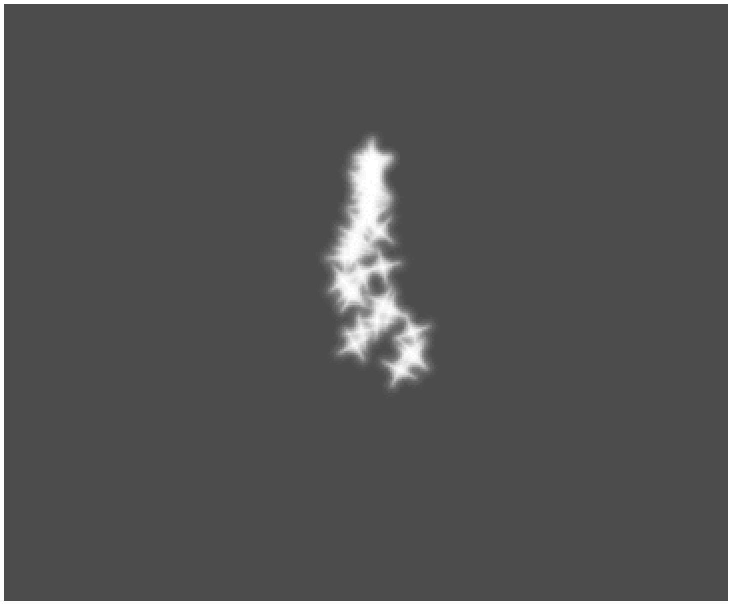
Initial velocity is the speed at which particles will be emitted (in pixels/sec). Speed might later be modified by gravity or other accelerations (as described further below).

Angular Velocity

Angular velocity is the initial angular velocity applied to particles.

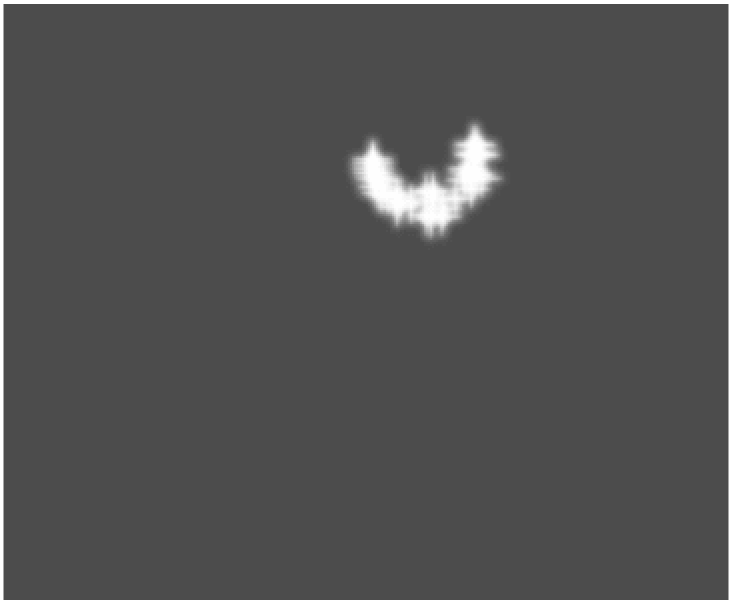
Spin Velocity

Spin velocity is the speed at which particles turn around their center (in degrees/sec).



Orbit Velocity

Orbit velocity is used to make particles turn around their center.



Linear Acceleration

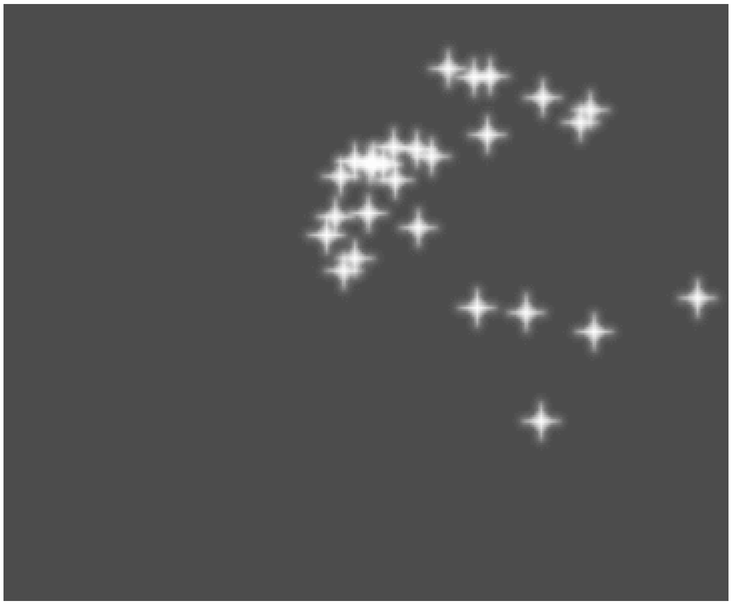
The linear acceleration applied to each particle.

Radial Acceleration

If this acceleration is positive, particles are accelerated away from the center. If negative, they are absorbed towards it.

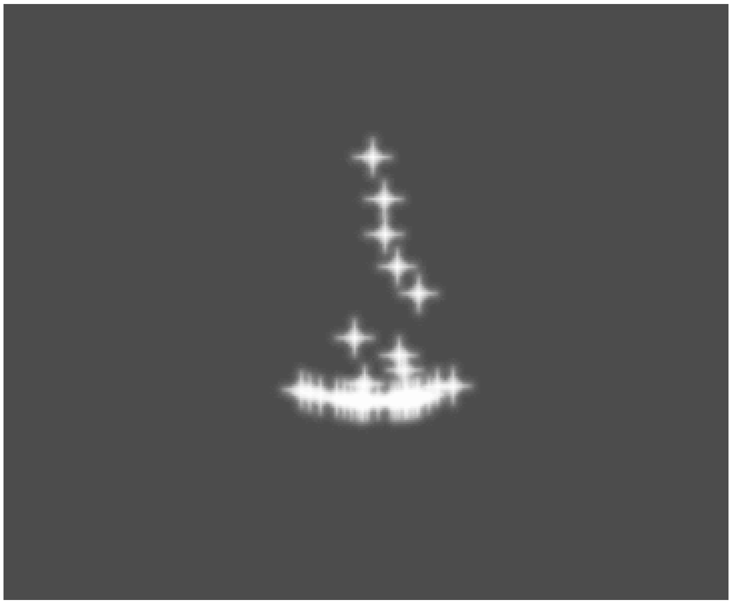
Tangential Acceleration

This acceleration will use the tangent vector to the center. Combining with radial acceleration can do nice effects.



Damping

Damping applies friction to the particles, forcing them to stop. It is especially useful for sparks or explosions, which usually begin with a high linear velocity and then stop as they fade.

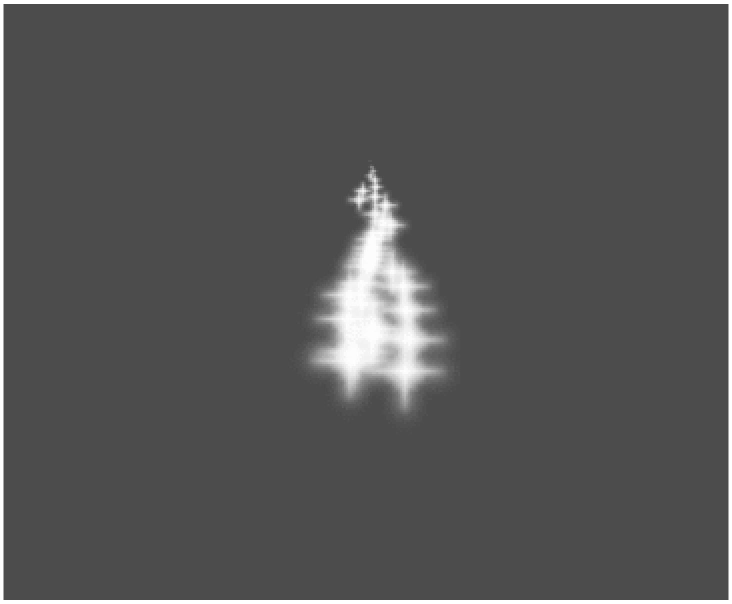


Angle

Determines the initial angle of the particle (in degrees). This parameter is mostly useful randomized.

Scale

Determines the initial scale of the particles.



Color

Used to change the color of the particles being emitted.

Hue Variation

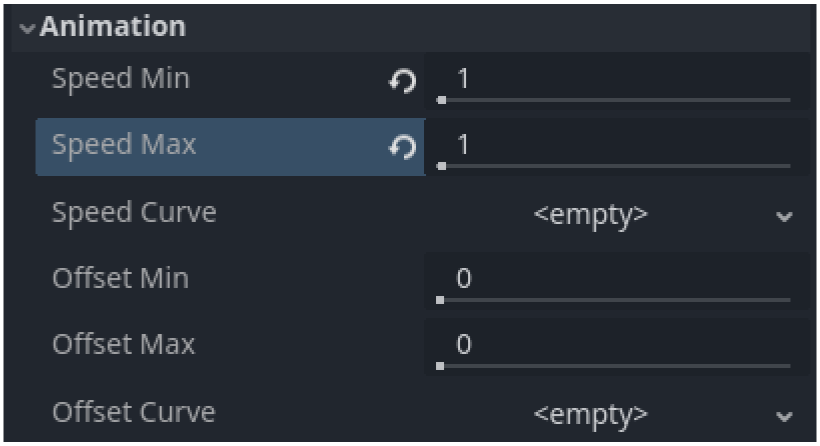
The `Variation` value sets the initial hue variation applied to each particle. The `Variation Random` value controls the hue variation randomness ratio.

Animation

Note

Particle flipbook animation is only effective if the CanvasItemMaterial used on the GPUParticles2D or CPUParticles2D node has been [configured accordingly](#).

To set up the particle flipbook for linear playback, set the **Speed Min** and **Speed Max** values to 1:



Speed Min: 1
Speed Max: 1
Speed Curve: <empty>
Offset Min: 0
Offset Max: 0
Offset Curve: <empty>

Setting up particle animation for playback during the particle's lifetime

By default, looping is disabled. If the particle is done playing before its lifetime ends, the particle will keep using the flipbook's last frame (which may be fully transparent depending on how the flipbook texture is designed). If looping is enabled, the animation will loop back to the first frame and resume playing.

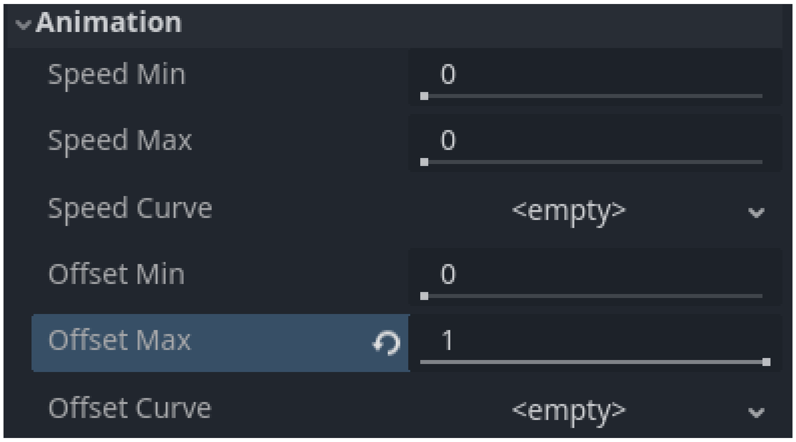
Depending on how many images your sprite sheet contains and for how long your particle is alive, the animation might not look smooth. The relationship between particle lifetime, animation speed, and number of images in the sprite sheet is this:

Note

At an animation speed of `1.0`, the animation will reach the last image in the sequence just as the particle's lifetime ends.

$$Animation\ FPS = \frac{Number\ of\ images}{Lifetime}$$

If you wish the particle flipbook to be used as a source of random particle textures for every particle, keep the speed values at 0 and set **Offset Max** to 1 instead:



Animation
Speed Min 0
Speed Max 0
Speed Curve <empty>
Offset Min 0
Offset Max 1
Offset Curve <empty>

Setting up particle animation for random offset on emission

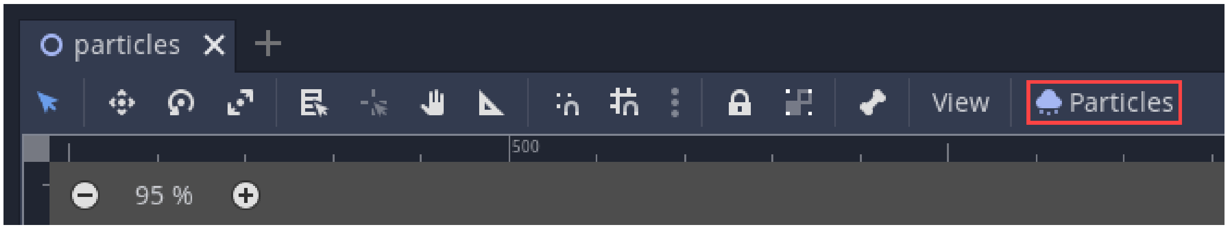
Note that the GPUParticles2D node's **Fixed FPS** also affects animation playback. For smooth animation playback, it's recommended to set it to 0 so that the particle is simulated on every rendered frame. If this is not an option for your use case, set **Fixed**

FPS to be equal to the effective framerate used by the flipbook animation (see above for the formula).

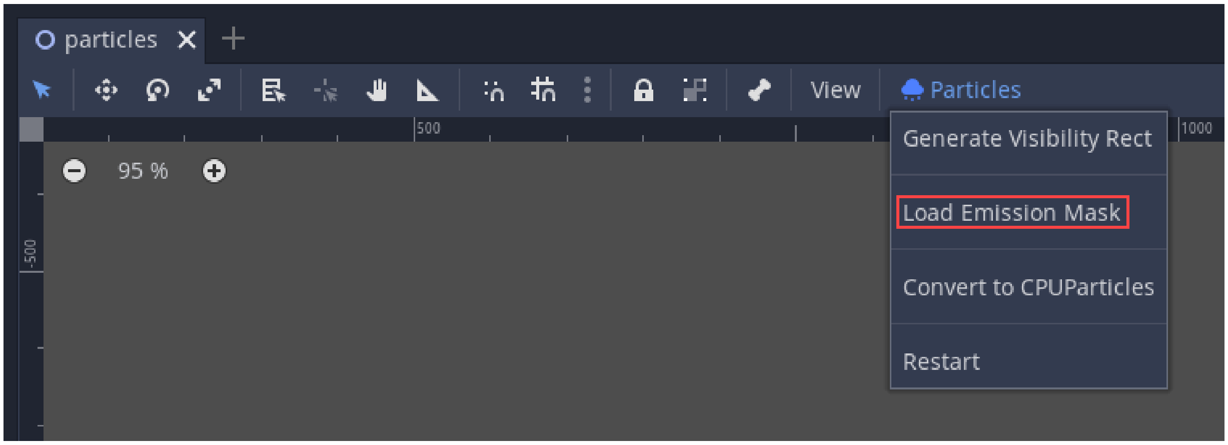
Emission Shapes

ParticleProcessMaterials allow you to set an Emission Mask, which dictates the area and direction in which particles are emitted. These can be generated from textures in your project.

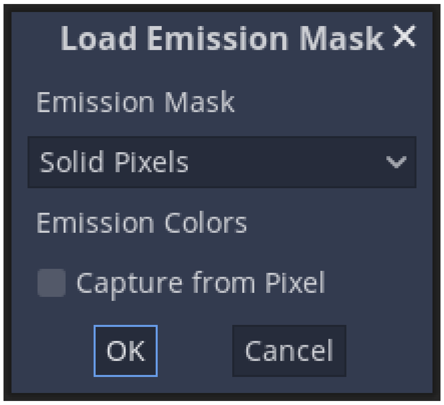
Ensure that a ParticleProcessMaterial is set, and the GPUParticles2D node is selected. A "Particles" menu should appear in the Toolbar:


particles
View
Particles
500
95 %

Open it and select "Load Emission Mask":


particles
View
Particles
500
1000
-500
95 %
Generate Visibility Rect
Load Emission Mask
Convert to CPUParticles
Restart

Then select which texture you want to use as your mask:


Load Emission Mask

Emission Mask: Solid Pixels

Emission Colors

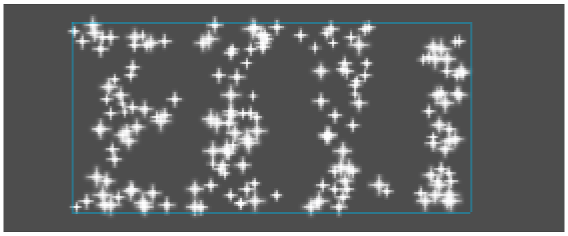
☐ Capture from Pixel

A dialog box with several settings will appear.

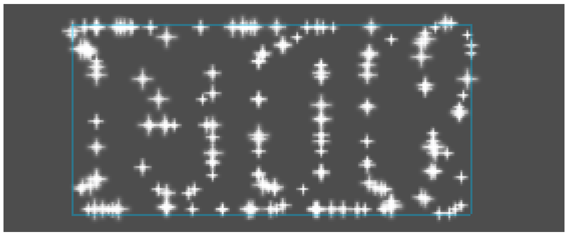
Emission Mask

Three types of emission masks can be generated from a texture:

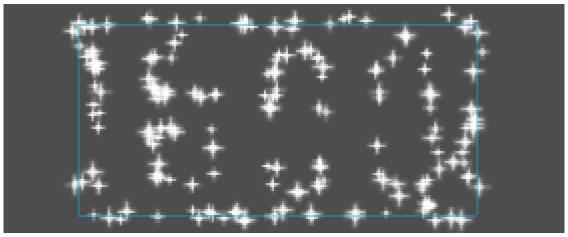
- Solid Pixels: Particles will spawn from any area of the texture, excluding transparent areas.



- Border Pixels: Particles will spawn from the outer edges of the texture.



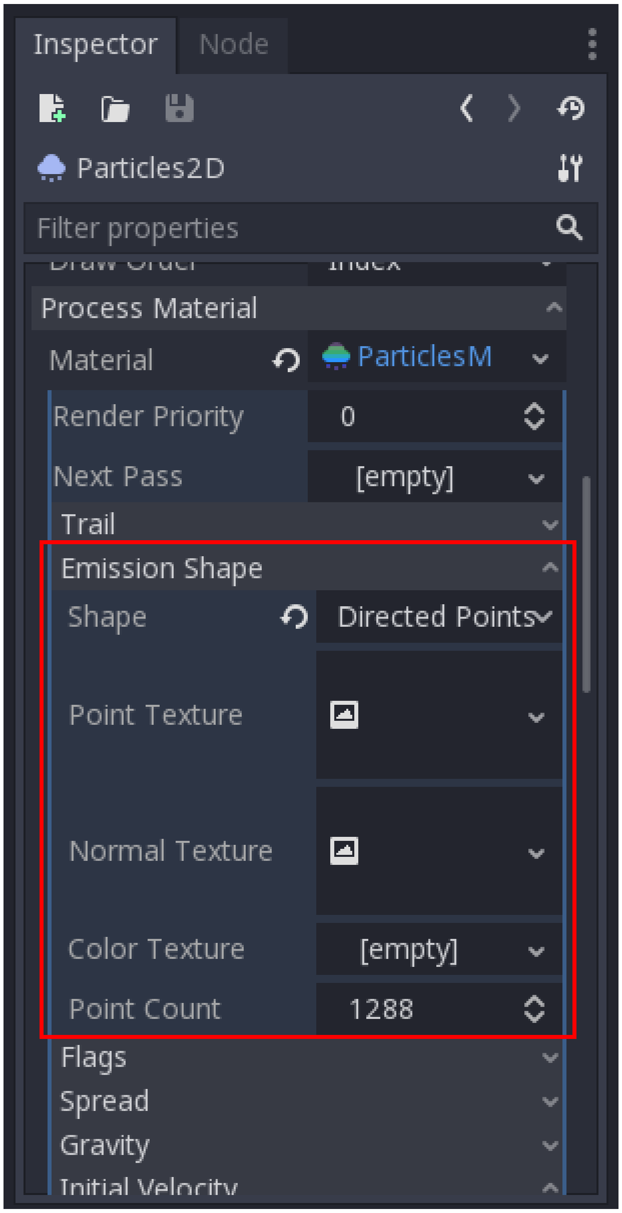
- Directed Border Pixels: Similar to Border Pixels, but adds extra information to the mask to give particles the ability to emit away from the borders. Note that an `Initial Velocity` will need to be set in order to utilize this.



Emission Colors

`Capture from Pixel` will cause the particles to inherit the color of the mask at their spawn points.

Once you click "OK", the mask will be generated and set to the ParticleProcessMaterial, under the `Emission Shape` section:



Inspector
Node
Particles2D
Filter properties
Draw Order
Index
Process Material

Material: ParticlesM

Render Priority: 0

Next Pass: [empty]

Trail
Emission Shape

Shape: Directed Points

Point Texture
Normal Texture

Color Texture: [empty]

Point Count: 1288

Flags
Spread
Gravity
Initial Velocity

All of the values within this section have been automatically generated by the "Load Emission Mask" menu, so they should generally be left alone.

Note

An image should not be added to `Point Texture` or `Color Texture` directly. The "Load Emission Mask" menu should always be used instead.

2D antialiasing

See also

Godot also supports antialiasing in 3D rendering. This is covered on the 3D antialiasing page.

Introduction

Due to their limited resolution, scenes rendered in 2D can exhibit aliasing artifacts. These artifacts usually manifest in the form of a "staircase" effect on geometry edges, and are most noticeable when using nodes such as Line2D, Polygon2D or TextureProgressBar. Custom drawing in 2D can also have aliasing artifacts for methods that don't support antialiasing.

In the example below, you can notice how edges have a blocky appearance:

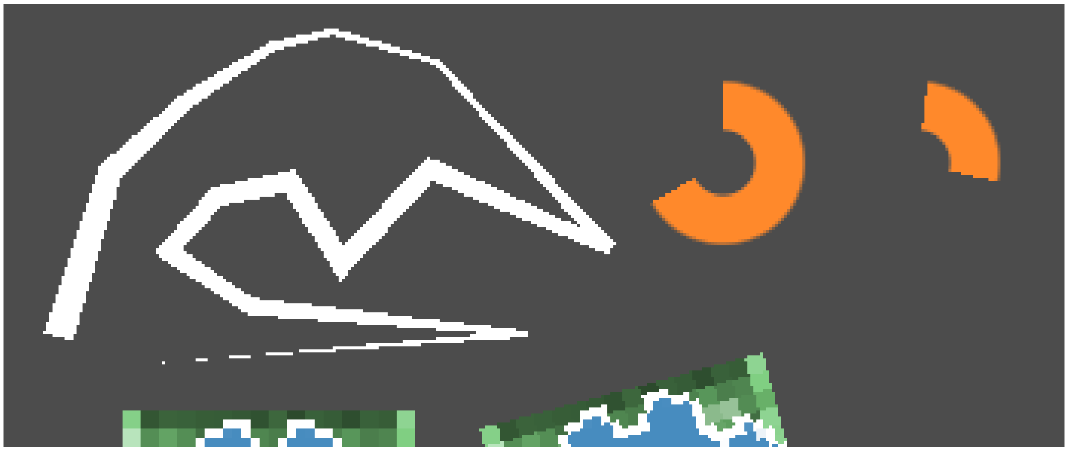


Image is scaled by 2× with nearest-neighbor filtering to make aliasing more noticeable.

To combat this, Godot supports several methods of enabling antialiasing on 2D rendering.

Antialiasing property in Line2D and custom drawing

This is the recommended method, as it has a lower performance impact in most cases.

Line2D has an **Antialiased** property which you can enable in the inspector. Also, several methods for Custom drawing in 2D support an optional `antialiased` parameter, which can be set to `true` when calling the function.

These methods do not require MSAA to be enabled, which makes their *baseline* performance cost low. In other words, there is no permanent added cost if you're not drawing any antialiased geometry at some point.

The downside of these antialiasing methods is that they work by generating additional geometry. If you're generating complex 2D geometry that's updated every frame, this may be a bottleneck. Also, Polygon2D, TextureProgressBar, and several custom drawing methods don't feature an antialiased property. For these nodes, you can use 2D multisample antialiasing instead.

Multisample antialiasing (MSAA)

Before enabling MSAA in 2D, it's important to understand what MSAA will operate on. MSAA in 2D follows similar restrictions as in 3D. While it does not introduce any blurriness, its scope of application is limited. The main applications of 2D MSAA are:

- Geometry edges, such as line and polygon drawing.

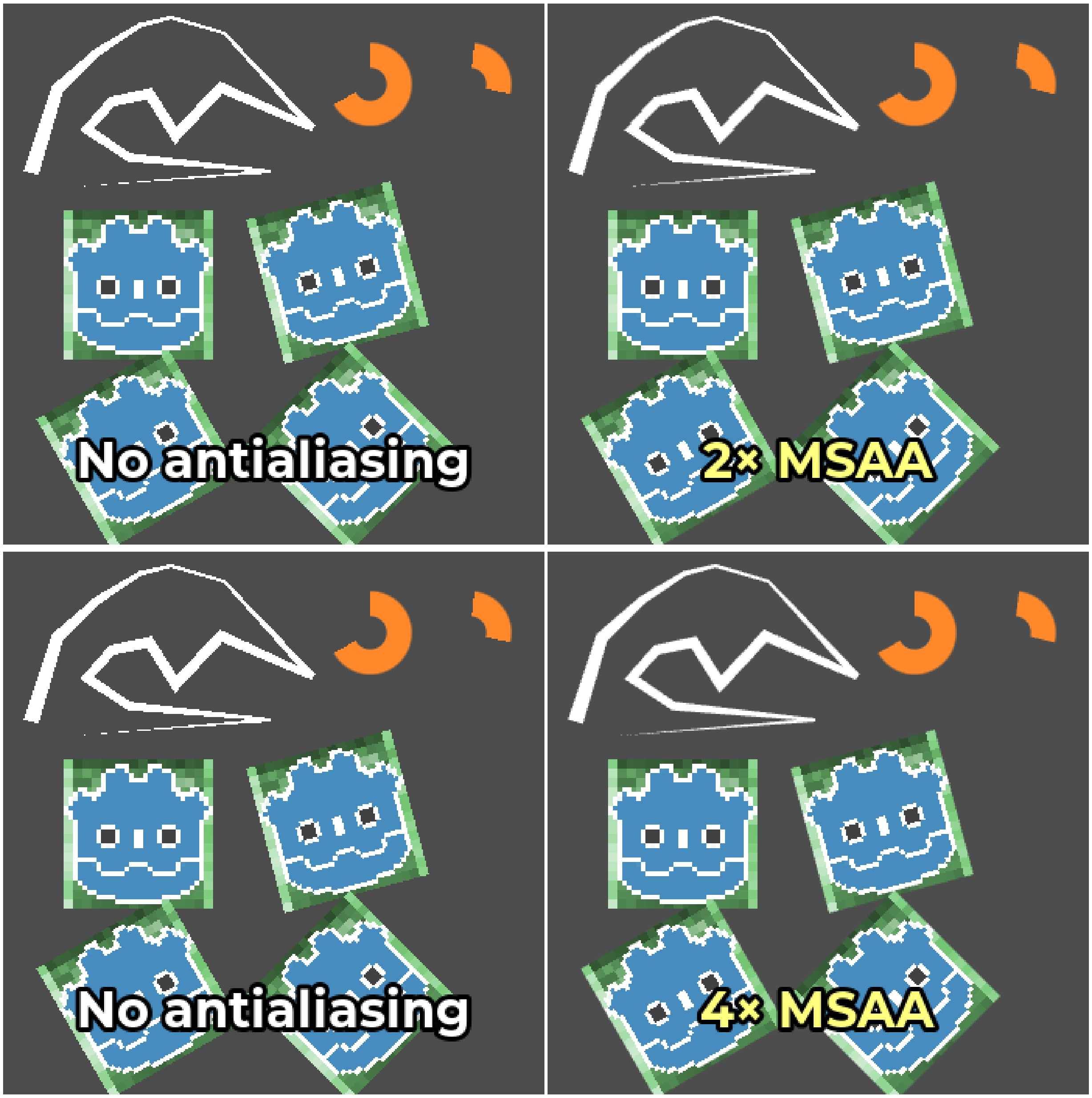
- Sprite edges *only for pixels touching one of the texture's edges*. This works for both linear and nearest-neighbor filtering. Sprite edges created using transparency on the image are not affected by MSAA.

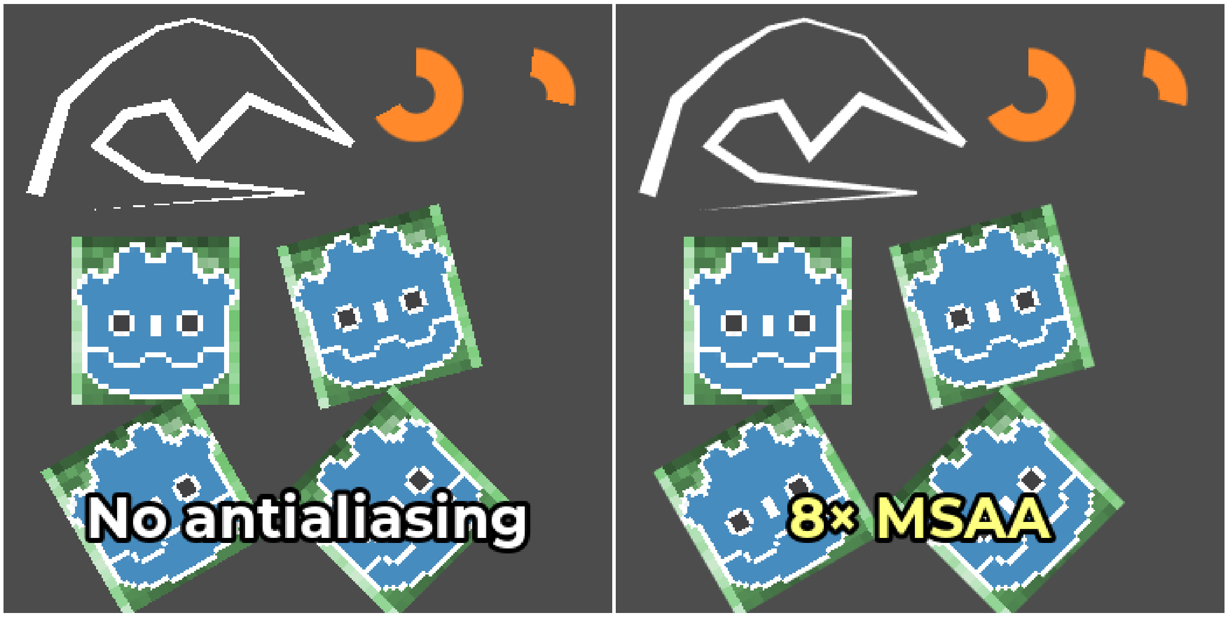
The downside of MSAA is that it only operates on edges. This is because MSAA increases the number of *coverage* samples, but not the number of *color* samples. However, since the number of color samples did not increase, fragment shaders are still run for each pixel only once. As a result, MSAA will **not affect** the following kinds of aliasing in any way:

- Aliasing *within* nearest-neighbor filtered textures (pixel art).
- Aliasing caused by custom 2D shaders.
- Specular aliasing when using Light2D.
- Aliasing in font rendering.

MSAA can be enabled in the Project Settings by changing the value of the **Rendering > Anti Aliasing > Quality > MSAA 2D** setting. It's important to change the value of the **MSAA 2D** setting and not **MSAA 3D**, as these are entirely separate settings.

Comparison between no antialiasing (left) and various MSAA levels (right). The top-left corner contains a Line2D node, the top-right corner contains 2 TextureProgressBar nodes. The bottom contains 8 pixel art sprites, with 4 of them touching the edges (green background) and 4 of them not touching the edges (Godot logo):


No antialiasing
2× MSAA
No antialiasing
4× MSAA


No antialiasing
8× MSAA

Custom drawing in 2D

Introduction

Godot has nodes to draw sprites, polygons, particles, text, and many other common game development needs. However, if you need something specific not covered with the standard nodes you can make any 2D node (for example, Control or Node2D-based) draw on screen using custom commands.

Custom drawing in a 2D node is *really* useful. Here are some use cases:

- Drawing shapes or logic that existing nodes can't do, such as an image with trails or a special animated polygon.
- Drawing a large number of simple objects, such as a grid or a board for a 2d game. Custom drawing avoids the overhead of using a large number of nodes, possibly lowering memory usage and improving performance.
- Making a custom UI control. There are plenty of controls available, but when you have unusual needs, you will likely need a custom control.

Drawing

Add a script to any CanvasItem derived node, like Control or Node2D. Then override the _draw() function.

GDScript

```
extends Node2D
```

```
func _draw():
    pass  # Your draw commands here.
```

C#

```
using Godot;

public partial class MyNode2D : Node2D
{
    public override void _Draw()
    {
        // Your draw commands here.
    }
}
```

Draw commands are described in the CanvasItem class reference. There are plenty of them and we will see some of them in the examples below.

Updating

The _draw function is only called once, and then the draw commands are cached and remembered, so further calls are unnecessary.

If re-drawing is required because a variable or something else changed, call CanvasItem.queue_redraw in that same node and a new `_draw()` call will happen.

Here is a little more complex example, where we have a texture variable that can be modified at any time, and using a setter, it forces a redraw of the texture when modified:

GDScript

```
extends Node2D
```

```
@export var texture : Texture2D:
    set(value):
        texture = value
        queue_redraw()

func _draw():
    draw_texture(texture, Vector2())
```

C#

```
using Godot;

public partial class MyNode2D : Node2D
{
    private Texture2D _texture;

    [Export]
    public Texture2D Texture
    {
        get
        {
            return _texture;
        }

        set
        {
            _texture = value;
            QueueRedraw();
        }
    }

    public override void _Draw()
    {
        DrawTexture(_texture, new Vector2());
    }
}
```

To see it in action, you can set the texture to be the Godot icon on the editor by dragging and dropping the default `icon.svg` from the `FileSystem` tab to the Texture property on the `Inspector` tab. When changing the `Texture` property value while the previous script is running, the texture will also change automatically.

In some cases, we may need to redraw every frame. For this, call queue_redraw from the _process method, like this:

GDScript

```
extends Node2D

func _draw():
    pass  # Your draw commands here.

func _process(_delta):
    queue_redraw()
```

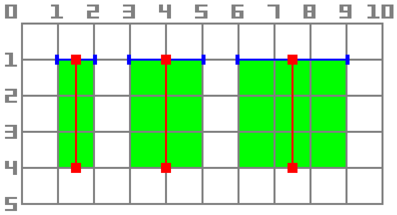
C#

```
using Godot;

public partial class MyNode2D : Node2D
{
    public override void _Draw()
    {
        // Your draw commands here.
    }

    public override void _Process(double delta)
    {
        QueueRedraw();
    }
}
```

Coordinates and line width alignment

The drawing API uses the CanvasItem's coordinate system, not necessarily pixel coordinates. This means `_draw()` uses the coordinate space created after applying the CanvasItem's transform. Additionally, you can apply a custom transform on top of it by using draw_set_transform or draw_set_transform_matrix.

When using draw_line, you should consider the width of the line. When using a width that is an odd size, the position of the start and end points should be shifted by `0.5` to keep the line centered, as shown below.



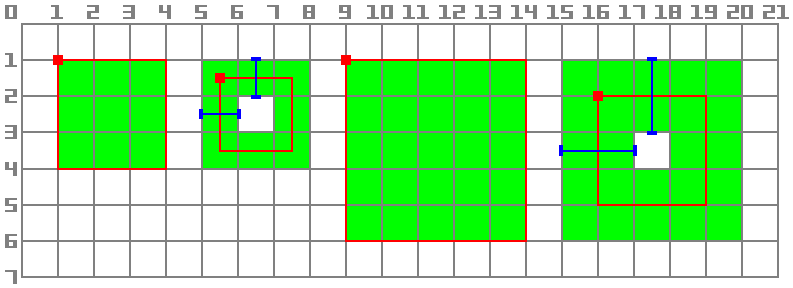
GDScript

```
func _draw():
    draw_line(Vector2(1.5, 1.0), Vector2(1.5, 4.0), Color.GREEN, 1.0)
    draw_line(Vector2(4.0, 1.0), Vector2(4.0, 4.0), Color.GREEN, 2.0)
    draw_line(Vector2(7.5, 1.0), Vector2(7.5, 4.0), Color.GREEN, 3.0)
```

C#

```
public override void _Draw()
{
    DrawLine(new Vector2(1.5f, 1.0f), new Vector2(1.5f, 4.0f), Colors.Green, 1.0f);
    DrawLine(new Vector2(4.0f, 1.0f), new Vector2(4.0f, 4.0f), Colors.Green, 2.0f);
    DrawLine(new Vector2(7.5f, 1.0f), new Vector2(7.5f, 4.0f), Colors.Green, 3.0f);
}
```

The same applies to the draw_rect method with `filled = false`.



GDScript

```
func _draw():
    draw_rect(Rect2(1.0, 1.0, 3.0, 3.0), Color.GREEN)
    draw_rect(Rect2(5.5, 1.5, 2.0, 2.0), Color.GREEN, false,
1.0)
    draw_rect(Rect2(9.0, 1.0, 5.0, 5.0), Color.GREEN)
    draw_rect(Rect2(16.0, 2.0, 3.0, 3.0), Color.GREEN, false,
2.0)
```

C#

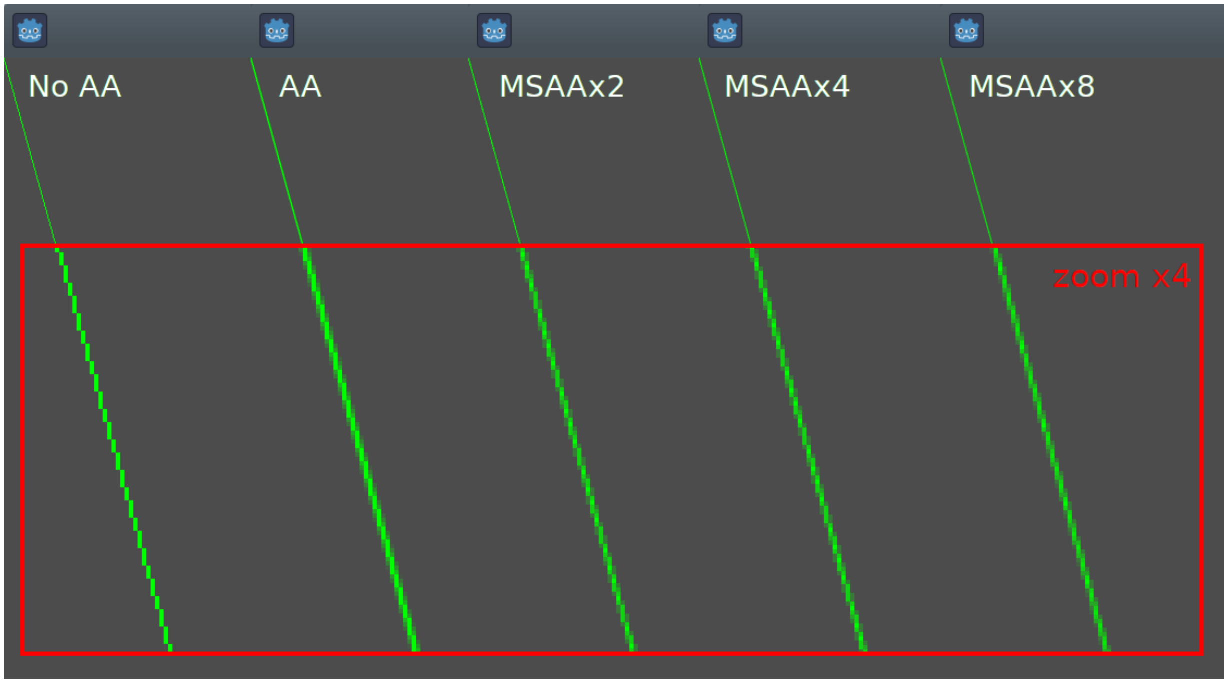
```
public override void _Draw()
{
    DrawRect(new Rect2(1.0f, 1.0f, 3.0f, 3.0f), Colors.Green);
    DrawRect(new Rect2(5.5f, 1.5f, 2.0f, 2.0f), Colors.Green,
false, 1.0f);
    DrawRect(new Rect2(9.0f, 1.0f, 5.0f, 5.0f), Colors.Green);
    DrawRect(new Rect2(16.0f, 2.0f, 3.0f, 3.0f), Colors.Green,
false, 2.0f);
}
```

Antialiased drawing

Godot offers method parameters in draw_line to enable antialiasing, but not all custom drawing methods offer this `antialiased` parameter.

For custom drawing methods that don't provide an `antialiased` parameter, you can enable 2D MSAA instead, which affects rendering in the entire viewport. This provides high-quality antialiasing, but a higher performance cost and only on specific elements. See [2D antialiasing] for more information.

Here is a comparison of a line of minimal width (`width=-1`) drawn with `antialiased=false`, `antialiased=true`, and `antialiased=false` with 2D MSAA 2x, 4x, and 8x enabled.



Tools

Drawing your own nodes might also be desired while running them in the editor. This can be used as a preview or visualization of some feature or behavior.

To do this, you can use the tool annotation on both GDScript and C#. See the example below and Running code in the editor for more information.

Example 1: drawing a custom shape

We will now use the custom drawing functionality of the Godot Engine to draw something that Godot doesn't provide functions for. We will recreate the Godot logo but with code- only using drawing functions.

You will have to code a function to perform this and draw it yourself.

Note

The following instructions use a fixed set of coordinates that could be too small for high resolution screens (larger than 1080p). If that is your case, and the drawing is too small consider increasing your window scale in `Menu > Project > Project settings > display/window/stretch/scale` to adjust the project to a higher resolution (a 2 or 4 scale tends to work well).

Drawing a custom polygon shape

While there is a dedicated node to draw custom polygons (Polygon2D), we will use in this case exclusively lower level drawing functions to combine them on the same node and be able to create more complex shapes later on.

First, we will define a set of points -or X and Y coordinates- that will form the base of our shape:

GDScript

```
extends Node2D

var coords_head : Array = [
    [ 22.952, 83.271 ],  [ 28.385, 98.623 ],
    [ 53.168, 107.647 ], [ 72.998, 107.647 ],
    [ 99.546, 98.623 ],  [ 105.048, 83.271 ],
    [ 105.029, 55.237 ], [ 110.740, 47.082 ],
    [ 102.364, 36.104 ], [ 94.050, 40.940 ],
    [ 85.189, 34.445 ],  [ 85.963, 24.194 ],
    [ 73.507, 19.930 ],  [ 68.883, 28.936 ],
    [ 59.118, 28.936 ],  [ 54.494, 19.930 ],
    [ 42.039, 24.194 ],  [ 42.814, 34.445 ],
    [ 33.951, 40.940 ],  [ 25.637, 36.104 ],
    [ 17.262, 47.082 ],  [ 22.973, 55.237 ]
]
```

C#

```
using Godot;

public partial class MyNode2D : Node2D
{
    private float[,] _coordsHead =
    {
        { 22.952f, 83.271f },  { 28.385f, 98.623f },
        { 53.168f, 107.647f }, { 72.998f, 107.647f },
        { 99.546f, 98.623f },  { 105.048f, 83.271f },
        { 105.029f, 55.237f }, { 110.740f, 47.082f },
        { 102.364f, 36.104f }, { 94.050f, 40.940f },
        { 85.189f, 34.445f },  { 85.963f, 24.194f },
        { 73.507f, 19.930f },  { 68.883f, 28.936f },
        { 59.118f, 28.936f },  { 54.494f, 19.930f },
        { 42.039f, 24.194f },  { 42.814f, 34.445f },
        { 33.951f, 40.940f },  { 25.637f, 36.104f },
        { 17.262f, 47.082f },  { 22.973f, 55.237f }
    };
}
```

This format, while compact, is not the one that Godot understands to draw a polygon. In a different scenario we could have to load these coordinates from a file or calculate the positions while the application is running, so some transformation may be needed.

To transform these coordinates into the right format, we will create a new method `float_array_to_Vector2Array()`. Then we will override the `_ready()` function, which Godot will call only once -at the start of the execution- to load those coordinates into a variable:

GDScript

```
var head : PackedVector2Array

func float_array_to_Vector2Array(coords : Array) ->
PackedVector2Array:
    # Convert the array of floats into a PackedVector2Array.
    var array : PackedVector2Array = []
    for coord in coords:
        array.append(Vector2(coord[0], coord[1]))
    return array

func _ready():
    head = float_array_to_Vector2Array(coords_head);
```

C#

```
private Vector2[] _head;

private Vector2[] FloatArrayToVector2Array(float[,] coords)
{
    // Convert the array of floats into an array of Vector2.
    int size = coords.GetUpperBound(0);
    Vector2[] array = new Vector2[size + 1];
    for (int i = 0; i <= size; i++)
    {
        array[i] = new Vector2(coords[i, 0], coords[i, 1]);
    }
    return array;
}

public override void _Ready()
{
    _head = FloatArrayToVector2Array(_coordsHead);
}
```

To finally draw our first shape, we will use the method draw_polygon and pass the points (as an array of Vector2 coordinates) and its color, like this:

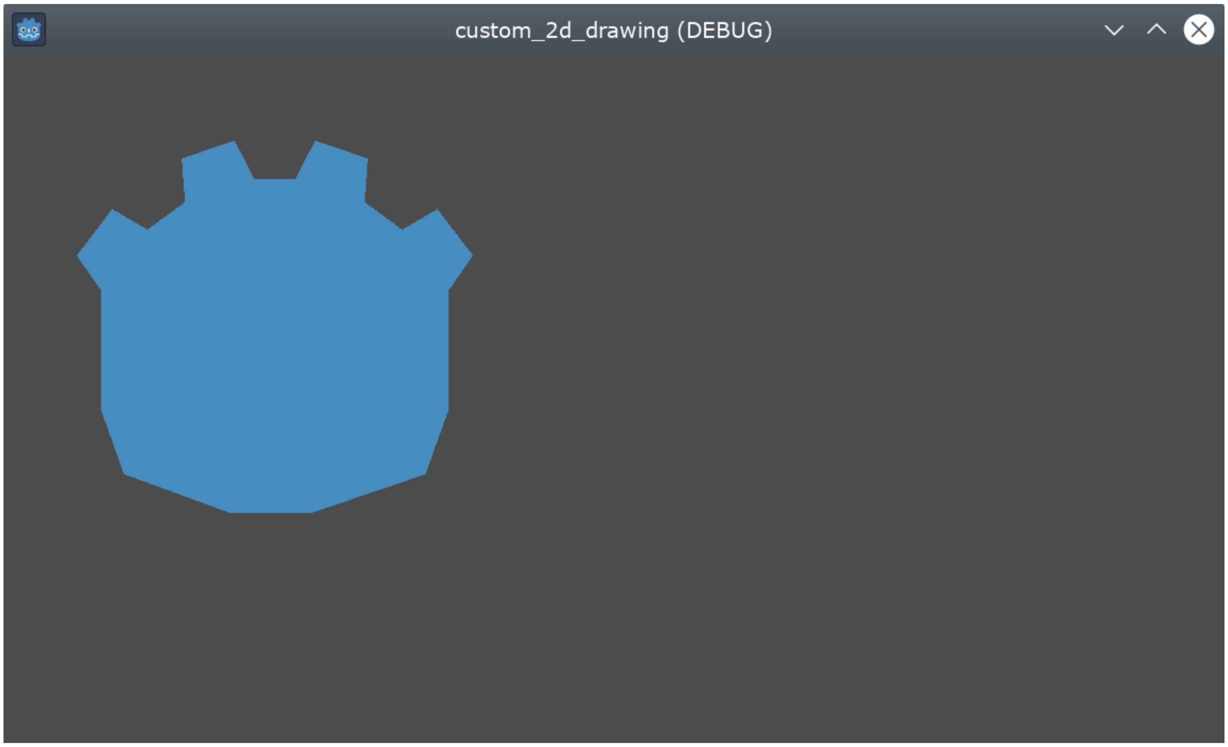
GDScript

```
func _draw():
    # We are going to paint with this color.
    var godot_blue : Color = Color("478cbf")
    # We pass the PackedVector2Array to draw the shape.
    draw_polygon(head, [ godot_blue ])
```

C#

```
public override void _Draw()
{
    // We are going to paint with this color.
    Color godotBlue = new Color("478cbf");
    // We pass the array of Vector2 to draw the shape.
    DrawPolygon(_head, new Color[]{ godotBlue });
}
```

When running it you should see something like this:



Note the lower part of the logo looks segmented- this is because a low amount of points were used to define that part. To simulate a smooth curve, we could add more points to our array, or maybe use a mathematical function to interpolate a curve and create a smooth shape from code (see example 2).

Polygons will always **connect its last defined point to its first one** in order to have a closed shape.

Drawing connected lines

Drawing a sequence of connected lines that don't close down to form a polygon is very similar to the previous method. We will use a connected set of lines to draw Godot's logo mouth.

First, we will define the list of coordinates that form the mouth shape, like this:

GDScript

```
var coords_mouth = [
    [ 22.817, 81.100 ], [ 38.522, 82.740 ],
    [ 39.001, 90.887 ], [ 54.465, 92.204 ],
    [ 55.641, 84.260 ], [ 72.418, 84.177 ],
    [ 73.629, 92.158 ], [ 88.895, 90.923 ],
    [ 89.556, 82.673 ], [ 105.005, 81.100 ]
]
```

C#

```
private float[,] _coordsMouth =
{
    { 22.817f, 81.100f }, { 38.522f, 82.740f },
    { 39.001f, 90.887f }, { 54.465f, 92.204f },
    { 55.641f, 84.260f }, { 72.418f, 84.177f },
    { 73.629f, 92.158f }, { 88.895f, 90.923f },
    { 89.556f, 82.673f }, { 105.005f, 81.100f }
};
```

We will load these coordinates into a variable and define an additional variable with the configurable line thickness:

GDScript

```
var mouth : PackedVector2Array
var _mouth_width : float = 4.4

func _ready():
    head = float_array_to_Vector2Array(coords_head);
    mouth = float_array_to_Vector2Array(coords_mouth);
```

C#

```
private Vector2[] _mouth;
private float _mouthWidth = 4.4f;

public override void _Ready()
{
    _head = FloatArrayToVector2Array(_coordsHead);
    _mouth = FloatArrayToVector2Array(_coordsMouth);
}
```

And finally we will use the method draw_polyline to actually draw the line, like this:

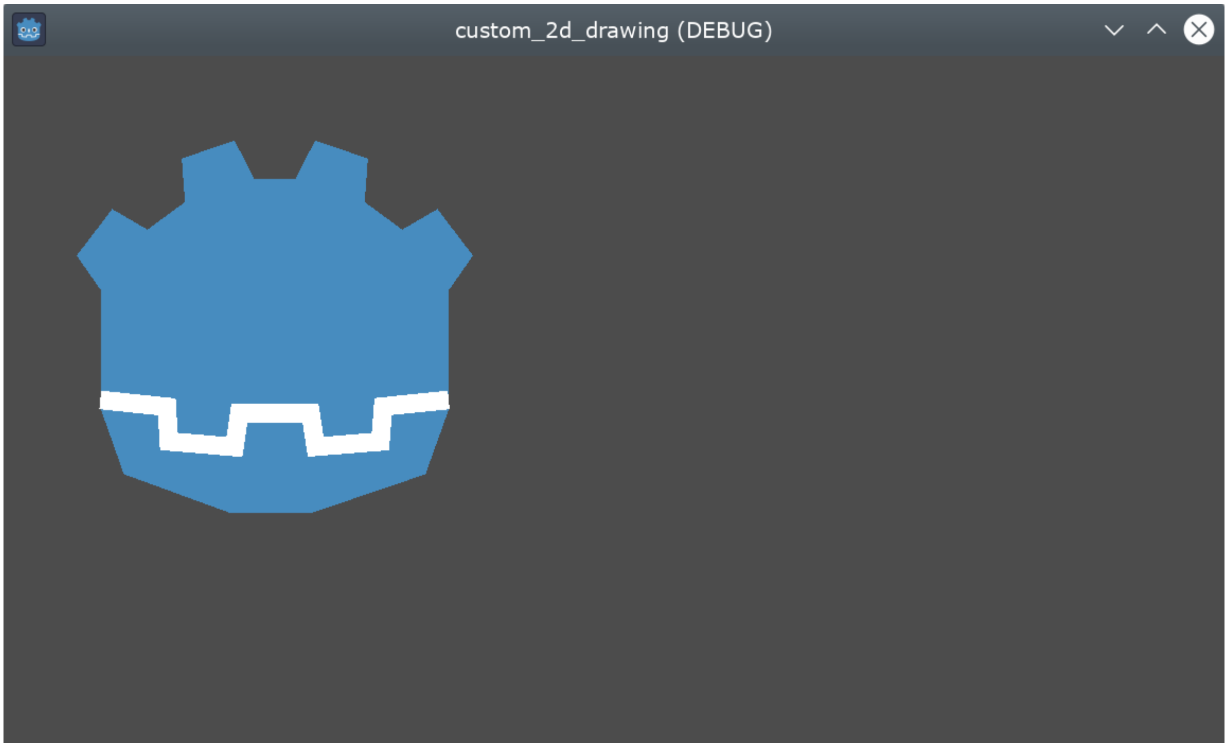
GDScript

```
func _draw():
    # We will use white to draw the line.
    var white : Color = Color.WHITE
    var godot_blue : Color = Color("478cbf")

    draw_polygon(head, [ godot_blue ])

    # We draw the while line on top of the previous shape.
    draw_polyline(mouth, white, _mouth_width)
```

C#

```
public override void _Draw()
{
    // We will use white to draw the line.
    Color white = Colors.White;
    Color godotBlue = new Color("478cbf");
    DrawPolygon(_head, new Color[]{ godotBlue });

    // We draw the while line on top of the previous shape.
    DrawPolyline(_mouth, white, _mouthWidth);
}
```

You should get the following output:



Unlike `draw_polygon()`, polylines can only have a single unique color for all its points (the second argument). This method has 2 additional arguments: the width of the line (which is as small as possible by default) and enabling or disabling the anti-aliasing (it is disabled by default).

The order of the `_draw` calls is important- like with the Node positions on the tree hierarchy, the different shapes will be drawn from top to bottom, resulting in the latest shapes hiding earlier ones if they overlap. In this case we want the mouth drawn over the head, so we put it afterwards.

Notice how we can define colors in different ways, either with a hexadecimal code or a predefined color name. Check the class Color for other constants and ways to define Colors.

Drawing circles

To create the eyes, we are going to add 4 additional calls to draw the eye shapes, in different sizes, colors and positions.

To draw a circle, you position it based on its center using the draw_circle method. The first parameter is a Vector2 with the coordinates of its center, the second is its radius, and the third is its color:

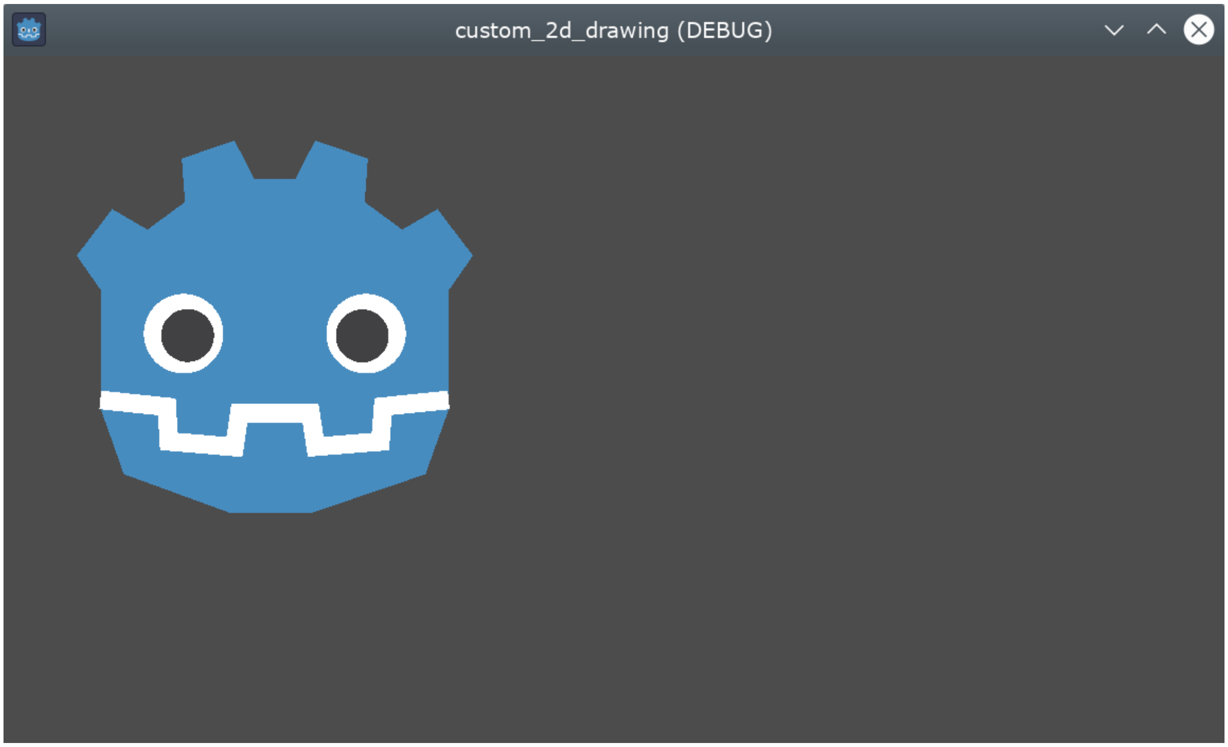
GDScript

```
func _draw():
    var white : Color = Color.WHITE
    var godot_blue : Color = Color("478cbf")
    var grey : Color = Color("414042")

    draw_polygon(head, [ godot_blue ])
    draw_polyline(mouth, white, _mouth_width)

    # Four circles for the 2 eyes: 2 white, 2 grey.
    draw_circle(Vector2(42.479, 65.4825), 9.3905, white)
    draw_circle(Vector2(85.524, 65.4825), 9.3905, white)
    draw_circle(Vector2(43.423, 65.92), 6.246, grey)
    draw_circle(Vector2(84.626, 66.008), 6.246, grey)
```

C#

```
public override void _Draw()
{
    Color white = Colors.White;
    Color godotBlue = new Color("478cbf");
    Color grey = new Color("414042");

    DrawPolygon(_head, new Color[]{ godotBlue });
    DrawPolyline(_mouth, white, _mouthWidth);

    // Four circles for the 2 eyes: 2 white, 2 grey.
    DrawCircle(new Vector2(42.479f, 65.4825f), 9.3905f, white);
    DrawCircle(new Vector2(85.524f, 65.4825f), 9.3905f, white);
    DrawCircle(new Vector2(43.423f, 65.92f), 6.246f, grey);
    DrawCircle(new Vector2(84.626f, 66.008f), 6.246f, grey);
}
```

When executing it, you should have something like this:



For partial, unfilled arcs (portions of a circle shape between certain arbitrary angles), you can use the method draw_arc.

Drawing lines

To draw the final shape (the nose) we will use a line to approximate it.

draw_line can be used to draw a single segment by providing its start and end coordinates as arguments, like this:

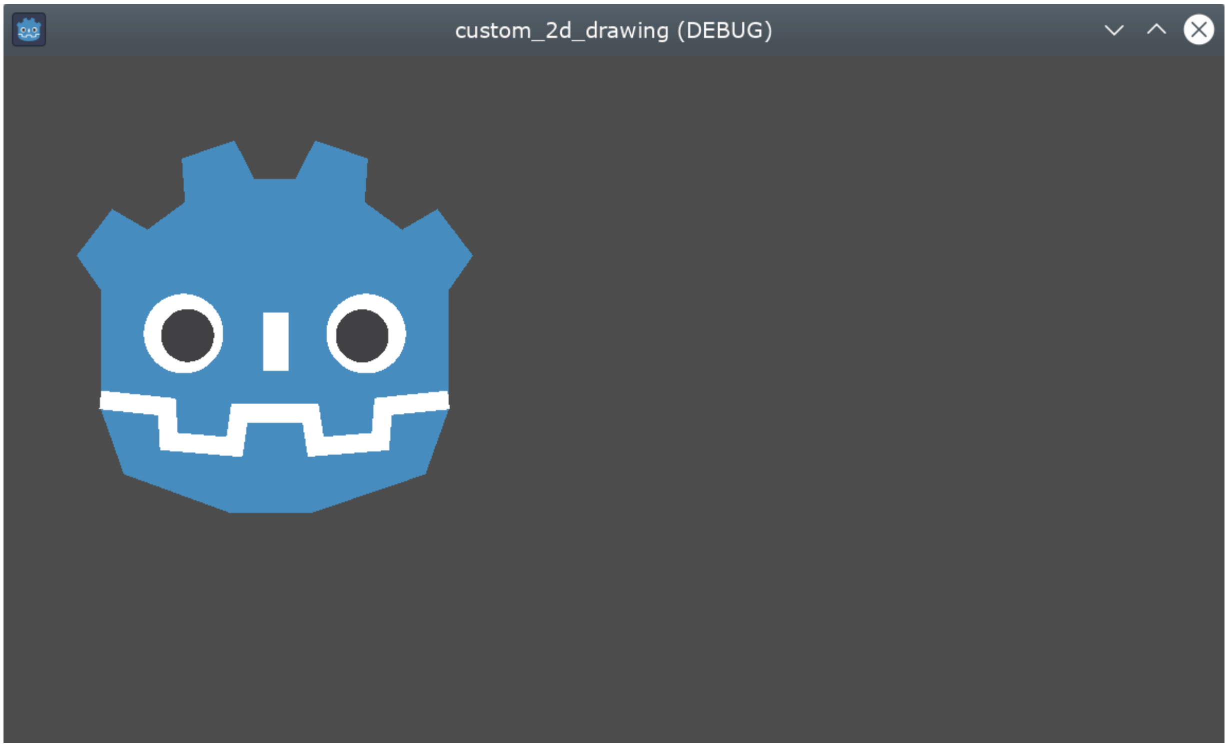
GDScript

```
func _draw():
    var white : Color = Color.WHITE
    var godot_blue : Color = Color("478cbf")
    var grey : Color = Color("414042")

    draw_polygon(head, [ godot_blue ])
    draw_polyline(mouth, white, _mouth_width)
    draw_circle(Vector2(42.479, 65.4825), 9.3905, white)
    draw_circle(Vector2(85.524, 65.4825), 9.3905, white)
    draw_circle(Vector2(43.423, 65.92), 6.246, grey)
    draw_circle(Vector2(84.626, 66.008), 6.246, grey)

    # Draw a short but thick white vertical line for the nose.
    draw_line(Vector2(64.273, 60.564), Vector2(64.273, 74.349),
white, 5.8)
```

C#

```
public override void _Draw()
{
    Color white = Colors.White;
    Color godotBlue = new Color("478cbf");
    Color grey = new Color("414042");

    DrawPolygon(_head, new Color[]{ godotBlue });
    DrawPolyline(_mouth, white, _mouthWidth);
    DrawCircle(new Vector2(42.479f, 65.4825f), 9.3905f, white);
    DrawCircle(new Vector2(85.524f, 65.4825f), 9.3905f, white);
    DrawCircle(new Vector2(43.423f, 65.92f), 6.246f, grey);
    DrawCircle(new Vector2(84.626f, 66.008f), 6.246f, grey);

    // Draw a short but thick white vertical line for the nose.
    DrawLine(new Vector2(64.273f, 60.564f), new
Vector2(64.273f, 74.349f),
             white, 5.8f);
}
```

You should now be able to see the following shape on screen:



Note that if multiple unconnected lines are going to be drawn at the same time, you may get additional performance by drawing all of them in a single call, using the draw_multiline method.

Drawing text

While using the Label Node is the most common way to add text to your application, the low level *_draw* function includes functionality to add text to your custom Node drawing. We will use it to add the name "GODOT" under the robot head.

We will use the draw_string method to do it, like this:

GDScript

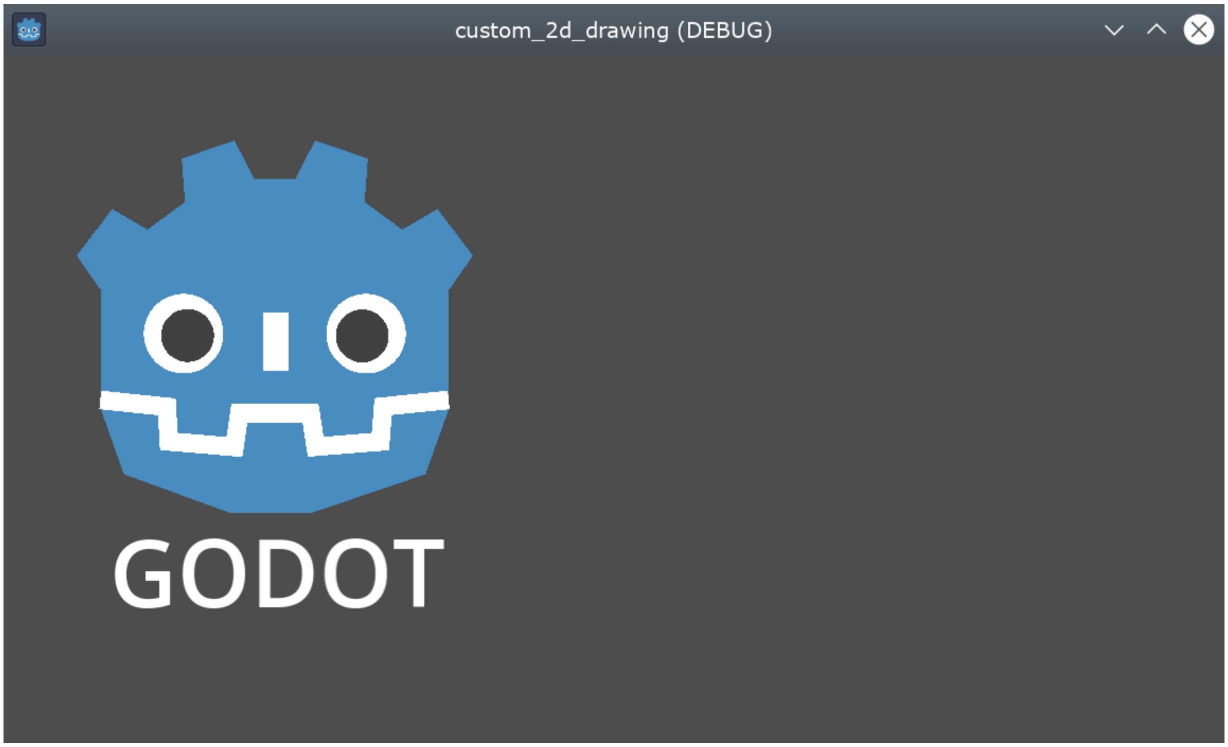
```
var default_font : Font = ThemeDB.fallback_font;

func _draw():
    var white : Color = Color.WHITE
    var godot_blue : Color = Color("478cbf")
    var grey : Color = Color("414042")

    draw_polygon(head, [ godot_blue ])
    draw_polyline(mouth, white, _mouth_width)
    draw_circle(Vector2(42.479, 65.4825), 9.3905, white)
    draw_circle(Vector2(85.524, 65.4825), 9.3905, white)
    draw_circle(Vector2(43.423, 65.92), 6.246, grey)
    draw_circle(Vector2(84.626, 66.008), 6.246, grey)
    draw_line(Vector2(64.273, 60.564), Vector2(64.273, 74.349),
white, 5.8)

    # Draw GODOT text below the logo with the default font,
size 22.
    draw_string(default_font, Vector2(20, 130), "GODOT",
                HORIZONTAL_ALIGNMENT_CENTER, 90, 22)
```

C#

```
private Font _defaultFont = ThemeDB.FallbackFont;

public override void _Draw()
{
    Color white = Colors.White;
    Color godotBlue = new Color("478cbf");
    Color grey = new Color("414042");

    DrawPolygon(_head, new Color[]{ godotBlue });
    DrawPolyline(_mouth, white, _mouthWidth);
    DrawCircle(new Vector2(42.479f, 65.4825f), 9.3905f, white);
    DrawCircle(new Vector2(85.524f, 65.4825f), 9.3905f, white);
    DrawCircle(new Vector2(43.423f, 65.92f), 6.246f, grey);
    DrawCircle(new Vector2(84.626f, 66.008f), 6.246f, grey);
    DrawLine(new Vector2(64.273f, 60.564f), new
Vector2(64.273f, 74.349f),
             white, 5.8f);

    // Draw GODOT text below the logo with the default font,
size 22.
    DrawString(_defaultFont, new Vector2(20f, 130f), "GODOT",
                    HorizontalAlignment.Center, 90, 22);
}
```

Here we first load into the defaultFont variable the configured default theme font (a custom one can be set instead) and then we pass the following parameters: font, position, text, horizontal alignment, width, and font size.

You should see the following on your screen:



Additional parameters as well as other methods related to text and characters can be found on the CanvasItem class reference.

Show the drawing while editing

While the code so far is able to draw the logo on a running window, it will not show up on the `2D view` on the editor. In certain cases you would also like to show your custom Node2D or control on the editor, to position and scale it appropriately, like most other nodes do.

To show the logo directly on the editor (without running it), you can use the [@tool](#) annotation to request the custom drawing of the node to also appear while editing, like this:

GDScript

```
@tool
extends Node2D
```

C#

```
using Godot;

[Tool]
public partial class MyNode2D : Node2D
```

You will need to save your scene, rebuild your project (for C# only) and reload the current scene manually at the menu option `Scene > Reload Saved Scene` to refresh the current node in the `2D` view the first time you add or remove the `@tool` annotation.

Animation

If we wanted to make the custom shape change at runtime, we could modify the methods called or its arguments at execution time, or apply a transform.

For example, if we want the custom shape we just designed to rotate, we could add the following variable and code to the `_ready` and `_process` methods:

GDScript

```
extends Node2D
```

```
@export var rotation_speed : float = 1  # In radians per second.

func _ready():
    rotation = 0
    ...

func _process(delta: float):
    rotation -= rotation_speed * delta
```

C#

```
[Export]
public float RotationSpeed { get; set; } = 1.0f;  // In radians per second.

public override void _Ready()
{
    Rotation = 0;
    ...
}

public override void _Process(double delta)
{
    Rotation -= RotationSpeed * (float)delta;
}
```

The problem with the above code is that because we have created the points approximately on a rectangle starting from the upper left corner, the `(0, 0)` coordinate and extending to the right and down, we see that the rotation is done using the top left corner as pivot. A position transform change on the node won't help us here, as the rotation transform is applied first.

While we could rewrite all of the points' coordinates to be centered around `(0, 0)`, including negative coordinates, that would be a lot of work.

One possible way to work around this is to use the lower level draw_set_transform method to fix this issue, translating all points in the CanvasItem's own space, and then moving it back to its original place with a regular node transform, either in the editor or in code, like this:

GDScript

```
func _ready():
    rotation = 0
    position = Vector2(60, 60)
    ...

func _draw():
    draw_set_transform(Vector2(-60, -60))
    ...
```

C#

```
public override void _Ready()
{
    Rotation = 0;
    Position = new Vector2(60, 60);
    ...
}

public override void _Draw()
{
    DrawSetTransform(new Vector2(-60.0f, -60.0f));
    ...
}
```

This is the result, rotating around a pivot now on `(60, 60)`:

tutorials/2d/img/draw_godot_rotation.png

If what we wanted to animate was a property inside the `_draw()` call, we must remember to call `queue_redraw()` to force a refresh, as otherwise it would not be updated on screen.

For example, this is how we can make the robot appear to open and close its mouth, by changing the width of its mouth line follow a sinusoidal (sin) curve:

GDScript

```
var _mouth_width : float = 4.4
var _max_width : float = 7
var _time : float = 0

func _process(delta : float):
    _time += delta
    _mouth_width = abs(sin(_time) * _max_width)
    queue_redraw()

func _draw():
    ...
    draw_polyline(mouth, white, _mouth_width)
    ...
```

C#

```
private float _mouthWidth = 4.4f;
private float _maxWidth = 7f;
private float _time = 0f;

public override void _Process(double delta)
{
    _time += (float)delta;
    _mouthWidth = Mathf.Abs(Mathf.Sin(_time) * _maxWidth);
    QueueRedraw();
}

public override void _Draw()
{
    ...
    DrawPolyline(_mouth, white, _mouthWidth);
    ...
}
```

It will look somewhat like this when run:

tutorials/2d/img/draw_godot_mouth_animation.png

Please note that `_mouth_width` is a user defined property like any other and it or any other used as a drawing argument can be animated using more standard and high level methods such as a Tween or an AnimationPlayer Node. The only difference is that a `queue_redraw()` call is needed to apply those changes so they get shown on screen.

Example 2: drawing a dynamic line

The previous example was useful to learn how to draw and modify nodes with custom shapes and animations. This could have some advantages, such as using exact coordinates and vectors for drawing, rather than bitmaps -which means they will scale well when transformed on screen. In some cases, similar results could be achieved composing higher level functionality with nodes such as sprites or AnimatedSprites loading SVG resources (which are also images defined with vectors) and the AnimationPlayer node.

In other cases that will not be possible because we will not know what the resulting graphical representation will be before running the code. Here we will see how to draw a dynamic line whose coordinates are not known beforehand, and are affected by the user's input.

Drawing a straight line between 2 points

Let's assume we want to draw a straight line between 2 points, the first one will be fixed on the upper left corner `(0, 0)` and the second will be defined by the cursor position on screen.

We could draw a dynamic line between those 2 points like this:

GDScript

```
extends Node2D

var point1 : Vector2 = Vector2(0, 0)
var width : int = 10
var color : Color = Color.GREEN

var _point2 : Vector2

func _process(_delta):
    var mouse_position = get_viewport().get_mouse_position()
    if mouse_position != _point2:
        _point2 = mouse_position
        queue_redraw()

func _draw():
    draw_line(point1, _point2, color, width)
```

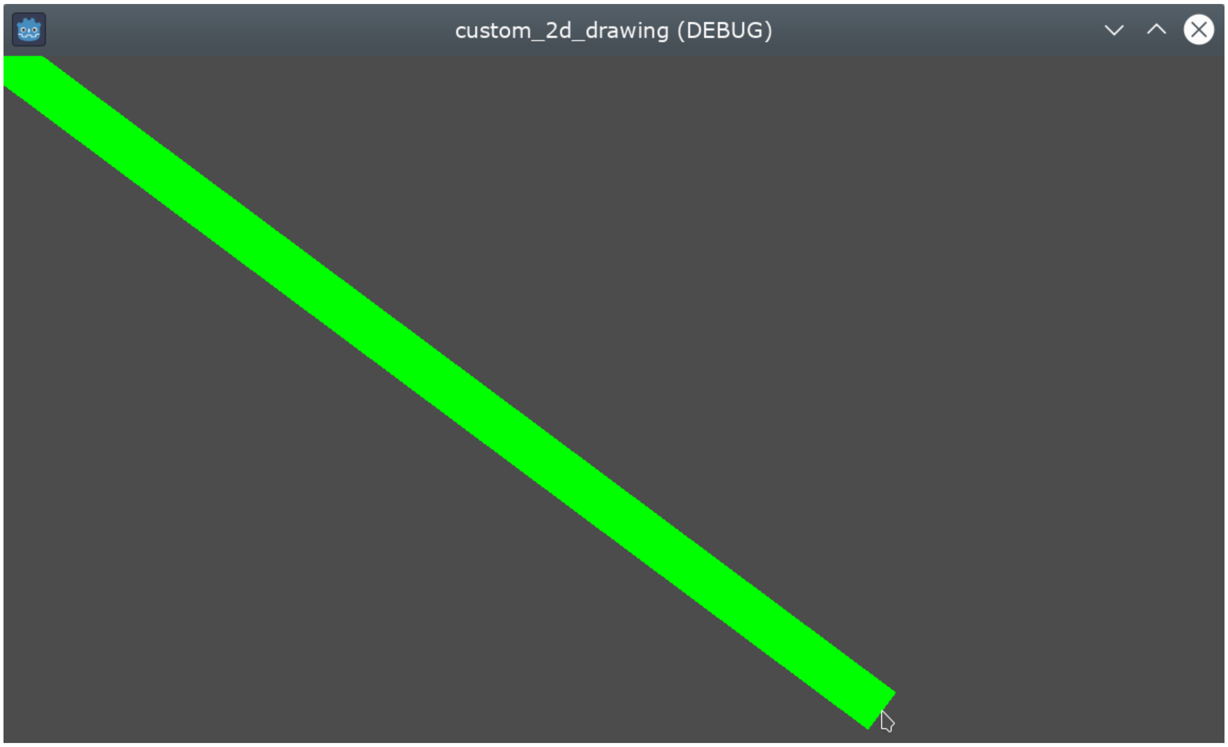
C#

```
using Godot;
using System;

public partial class MyNode2DLine : Node2D
{
    public Vector2 Point1 { get; set; } = new Vector2(0f, 0f);
    public int Width { get; set; } = 10;
    public Color Color { get; set; } = Colors.Green;

    private Vector2 _point2;

    public override void _Process(double delta)
    {
        Vector2 mousePosition =
GetViewport().GetMousePosition();
        if (mousePosition != _point2)
        {
            _point2 = mousePosition;
            QueueRedraw();
        }
    }

    public override void _Draw()
    {
        DrawLine(Point1, _point2, Color, Width);
    }
}
```

In this example we obtain the position of the mouse in the default viewport every frame with the method get_mouse_position. If the position has changed since the last draw request (a small optimization to avoid redrawing on every frame)- we will schedule a redraw. Our `_draw()` method only has one line: requesting the drawing of a green line of width 10 pixels between the top left corner and that obtained position.

The width, color, and position of the starting point can be configured with with the corresponding properties.

It should look like this when run:



Drawing an arc between 2 points

The above example works, but we may want to join those 2 points with a different shape or function, other than a straight line.

Let's try now creating an arc (a portion of a circumference) between both points.

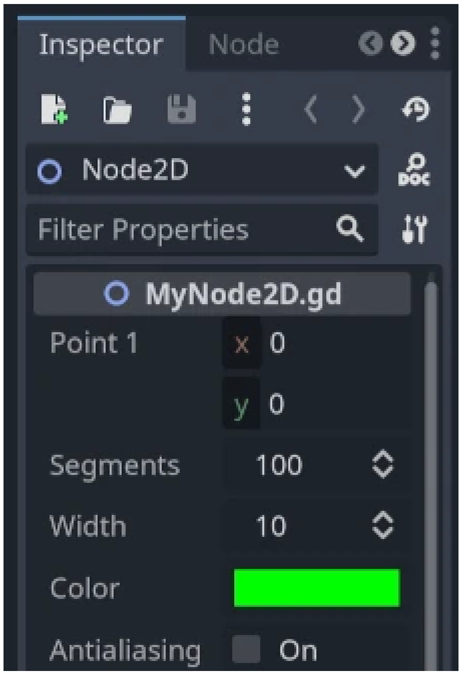
Exporting the line starting point, segments, width, color, and antialiasing will allow us to modify those properties very easily directly from the editor inspector panel:

GDScript

```
extends Node2D

@export var point1 : Vector2 = Vector2(0, 0)
@export_range(1, 1000) var segments : int = 100
@export var width : int = 10
@export var color : Color = Color.GREEN
@export var antialiasing : bool = false

var _point2 : Vector2
```

C#

```
using Godot;
using System;

public partial class MyNode2DLine : Node2D
{
    [Export]
    public Vector2 Point1 { get; set; } = new Vector2(0f, 0f);
    [Export]
    public float Length { get; set; } = 350f;
    [Export(PropertyHint.Range, "1,1000,")]
    public int Segments { get; set; } = 100;
    [Export]
    public int Width { get; set; } = 10;
    [Export]
    public Color Color { get; set; } = Colors.Green;
    [Export]
    public bool AntiAliasing { get; set; } = false;
    private Vector2 _point2;
}
```



Point 1 x 0 y 0

Segments 100

Width 10

Color 🟩

Antialiasing ☐ On

To draw the arc, we can use the method [draw_arc](). There are many arcs that pass through 2 points, so we will chose for this example the semicircle that has its center in the middle point between the 2 initial points.

Calculating this arc will be more complex than in the case of the line:

GDScript

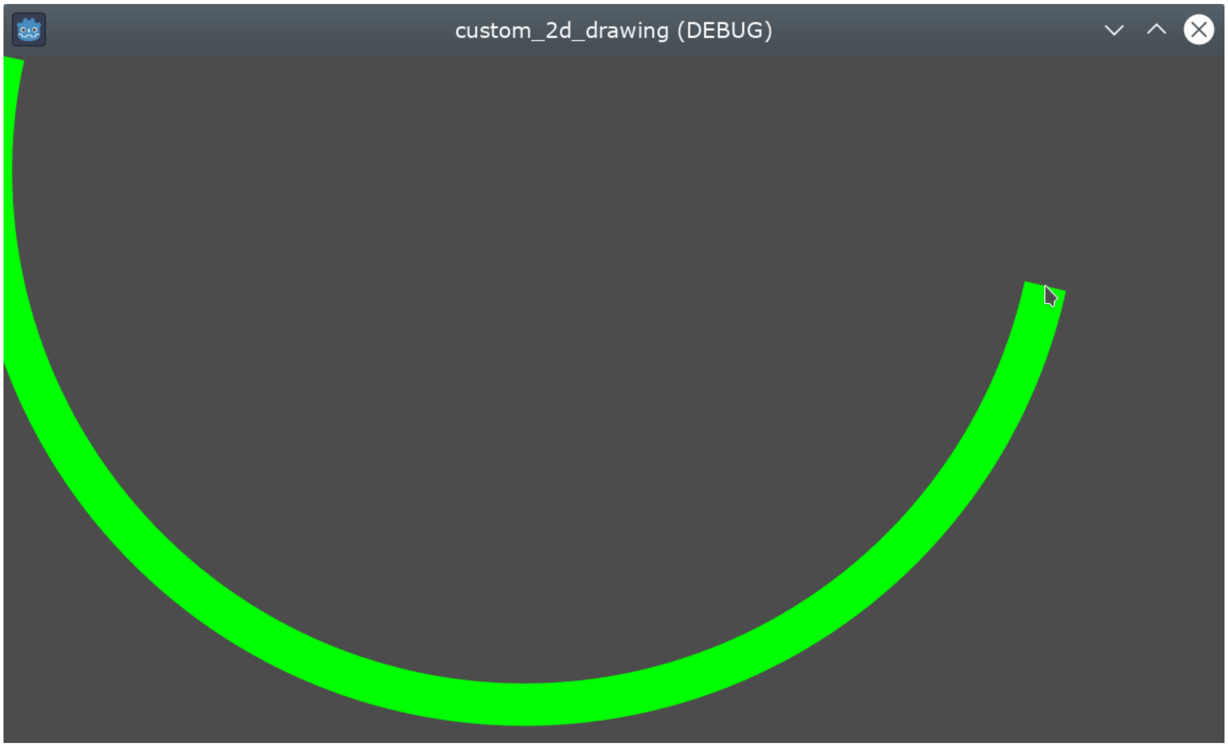
```
func _draw():
    # Calculate the arc parameters.
    var center : Vector2 = Vector2((_point2.x - point1.x) / 2,
                                   (_point2.y - point1.y) / 2)
    var radius : float = point1.distance_to(_point2) / 2
    var start_angle : float = (_point2 - point1).angle()
    var end_angle : float = (point1 - _point2).angle()
    if end_angle < 0:  # end_angle is likely negative,
normalize it.
        end_angle += TAU
    # Finally, draw the arc.
    draw_arc(center, radius, start_angle, end_angle, segments,
color,
             width, antialiasing)
```

C#

```
public override void _Draw()
{
    // Calculate the arc parameters.
    Vector2 center = new Vector2((_point2.X - Point1.X) / 2.0f,
                                    (_point2.Y - Point1.Y) /
2.0f);
    float radius = Point1.DistanceTo(_point2) / 2.0f;
    float startAngle = (_point2 - Point1).Angle();
    float endAngle = (Point1 - _point2).Angle();
    if (endAngle < 0.0f)  // endAngle is likely negative,
normalize it.
    {
        endAngle += Mathf.Tau;
    }

    // Finally, draw the arc.
    DrawArc(center, radius, startAngle, endAngle, Segments,
Color,
            Width, AntiAliasing);
}
```

The center of the semicircle will be the middle point between both points. The radius will be half the distance between both points. The start and end angles will be the angles of the vector from point1 to point2 and vice-versa. Note we had to normalize the `end_angle` in positive values because if `end_angle` is less than `start_angle`, the arc will be drawn counter-clockwise, which we don't want in this case (the arc would be upside-down).

The result should be something like this, with the arc going down and between the points:



Feel free to play with the parameters in the inspector to obtain different results: change the color, the width, the antialiasing, and increase the number of segments to increase the curve smoothness, at the cost of extra performance.

2D Parallax

Introduction

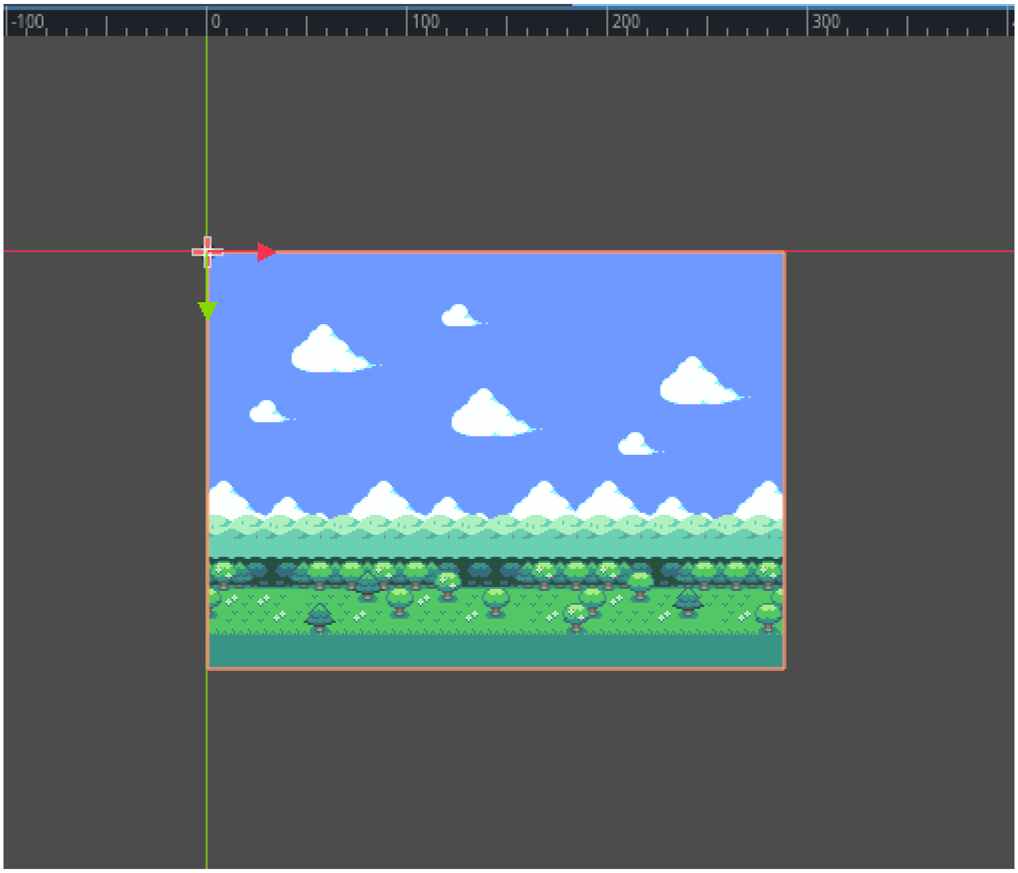
Parallax is an effect used to simulate depth by having textures move at different speeds relative to the camera. Godot provides the Parallax2D node to achieve this effect. It can still be easy to get tripped up though, so this page provides in-depth descriptions of some properties and how to fix some common mistakes.

Note

This page only covers how to use Parallax2D. This node is still experimental, so the implementation might change in future versions of Godot. However, it is still recommended to use over the ParallaxLayer and ParallaxBackground nodes.

Scroll scale

The backbone of the parallax effect is the scroll_scale property. It works as a scroll-speed multiplier, allowing layers to move at a different speed than the camera for each axis set. A value of 1 makes the parallax node scroll at the same speed as the camera. If you want your image to look further away when scrolling, use a value lower than 1, with 0 bringing it to a complete stop. If you want something to appear closer to the camera, use a value higher than 1, making it scroll faster.



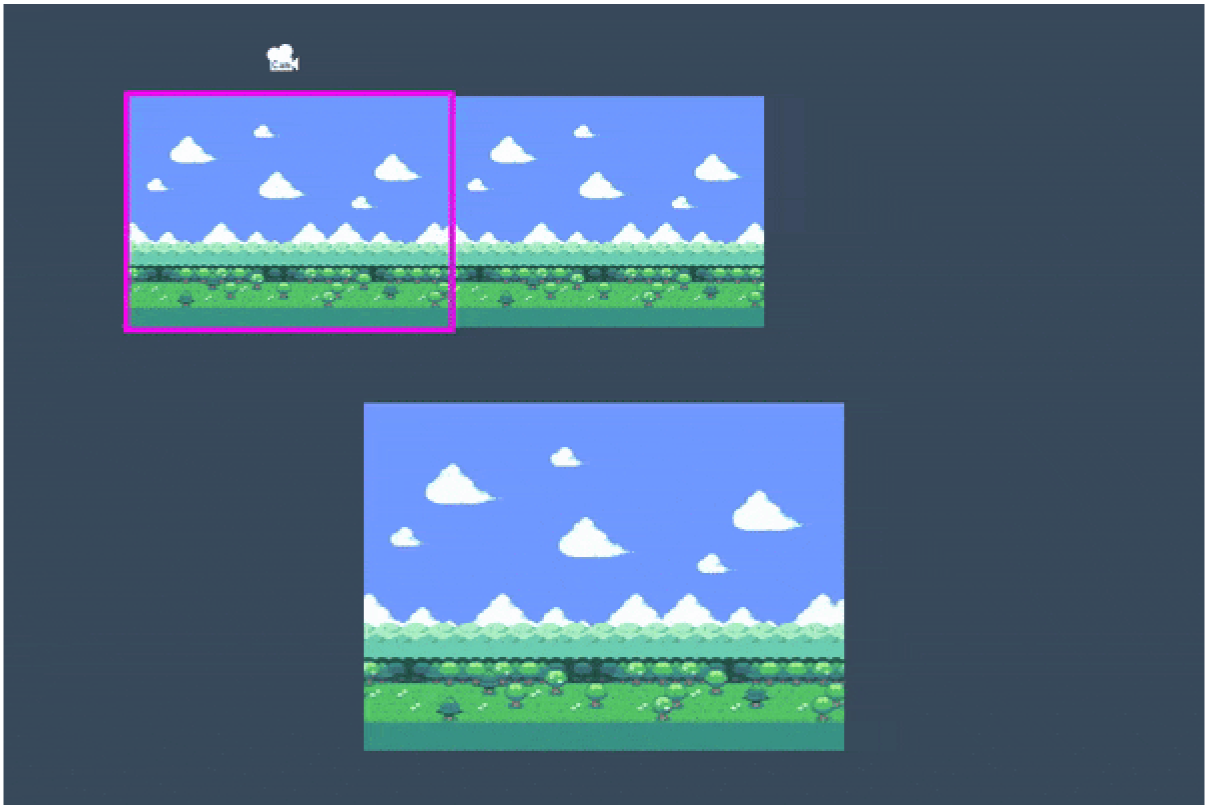
The scene above is comprised of five layers. Some good [scroll_scale](#) values might be:

- `(0.7, 1)` - Forest
- `(0.5, 1)` - Hills
- `(0.3, 1)` - Lower Clouds
- `(0.2, 1)` - Higher Clouds
- `(0.1, 1)` - Sky

The video below displays how these values affect scrolling while in-game:

Infinite repeat

Parallax2D provides a bonus effect that gives textures the illusion of repeating infinitely. repeat_size tells the node to snap its position forward or back when the camera scrolls by the set value. This effect is achieved by adding a single repeat to all the child canvas items offset by the value. While the camera scrolls between the image and its repeat, it invisibly snaps back giving the appearance of a looping image.

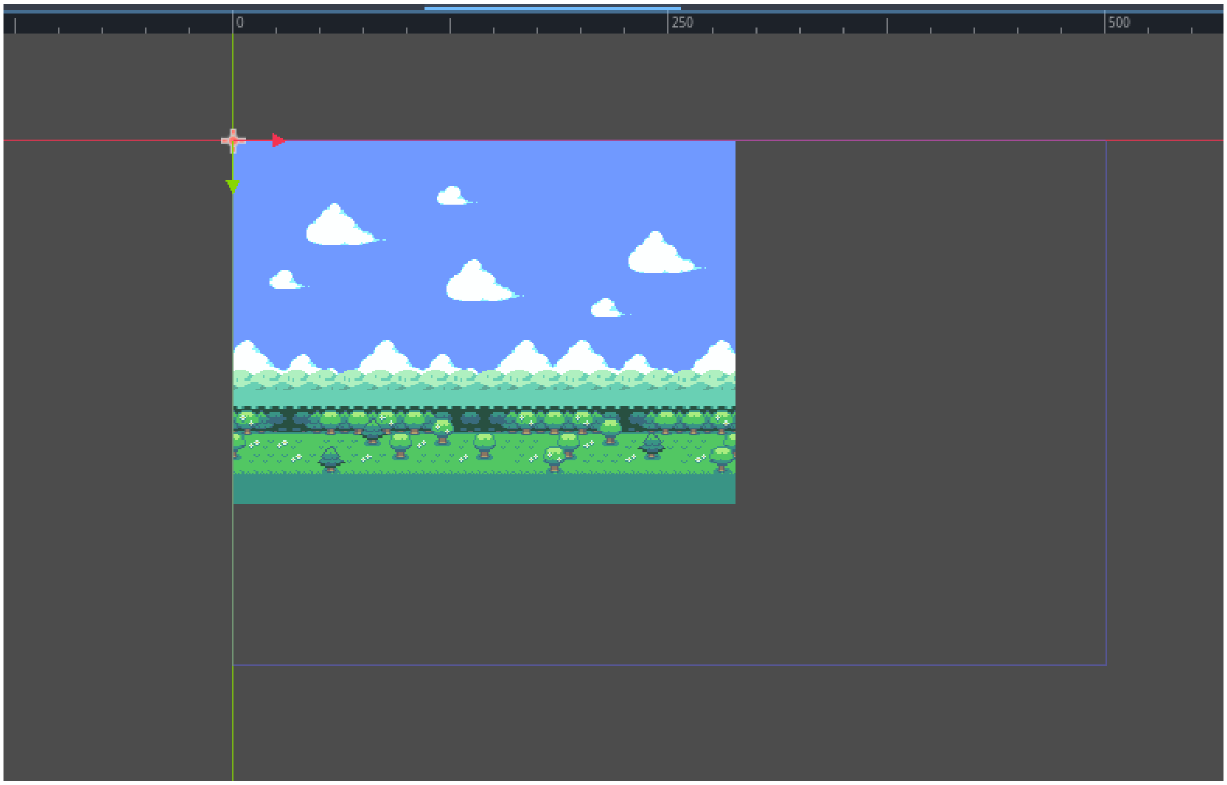


Being a delicate effect, it's easy for unfamiliar users to make mistakes with their setup. Let's go over the "how" and "why" of a few common problems users encounter.

Poor sizing

The infinite repeat effect is easiest to work with when you have an image designed to repeat seamlessly and is the same size or larger than your viewport **before** setting the repeat_size. If you aren't able to obtain assets that are designed for this task, there are some other things you can do to better prepare your image in regards to size.

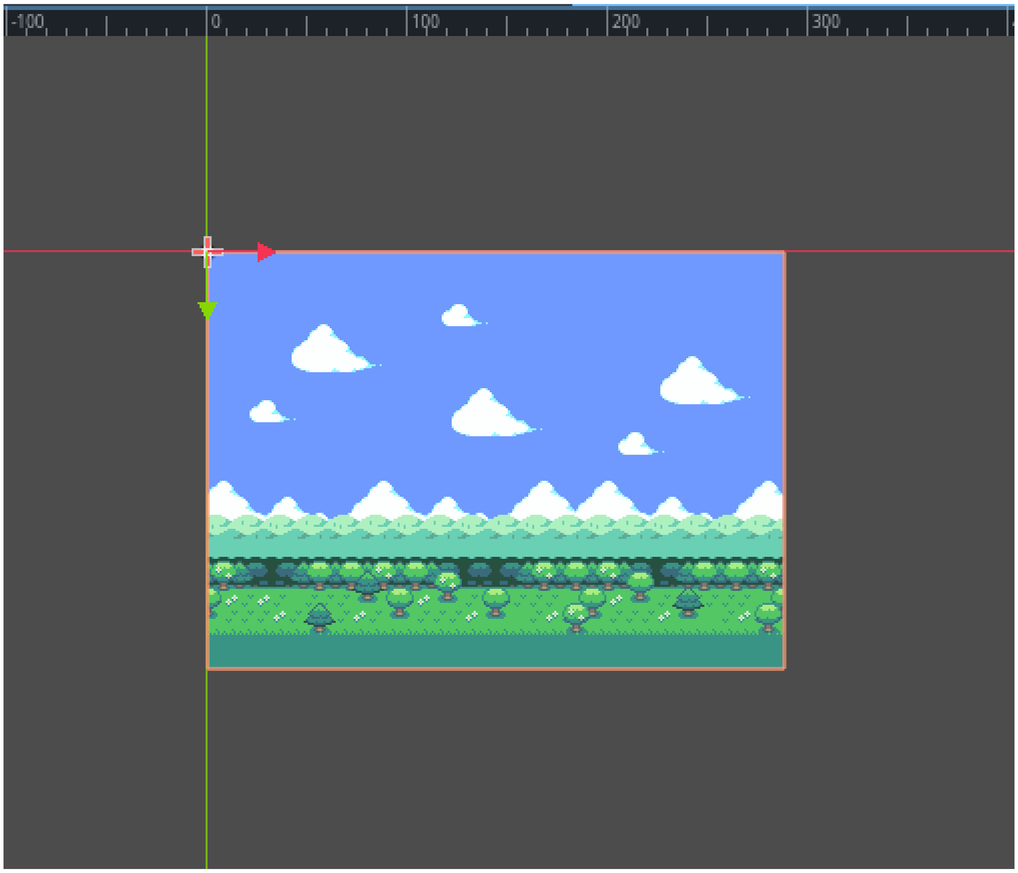
Here is an example of a texture that is too small for its viewport:



We can see that the viewport size is 500x300 but the texture is 288x208. If we set the [repeat_size](#) to the size of our image, the infinite repeat effect doesn't scroll properly because the original texture doesn't cover the viewport. If we set the [repeat_size](#) to the size of the viewport, we have a large gap. What can we do?

Make the viewport smaller

The simplest answer is to make the viewport the same size or smaller than your textures. Click on `Project -> Project Settings -> Window` and change the viewport height and width to match your background.

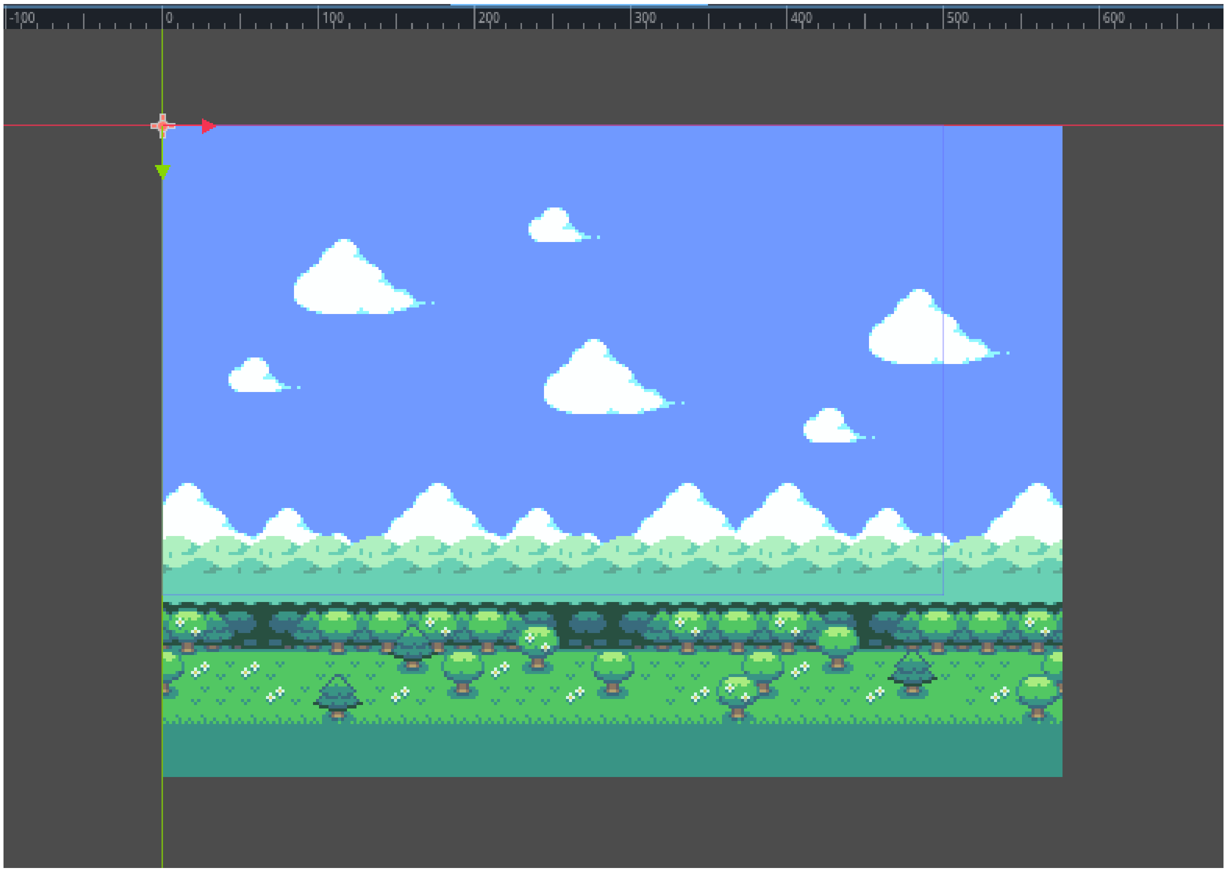


Scale the Parallax2D

If you're not aiming for a pixel-perfect style, or don't mind a little blurriness, you may opt to scale the textures larger to fit your screen. Set the scale of the Parallax2D, and all child textures scale with it.

Scale the child nodes

Similar to scaling the Parallax2D, you can scale your Sprite2D nodes to be large enough to cover the screen. Keep in mind that some settings like Parallax2D.repeat_size and Sprite2D.region_rect do not take scaling into account, so it's necessary to adjust these values based on the scale.

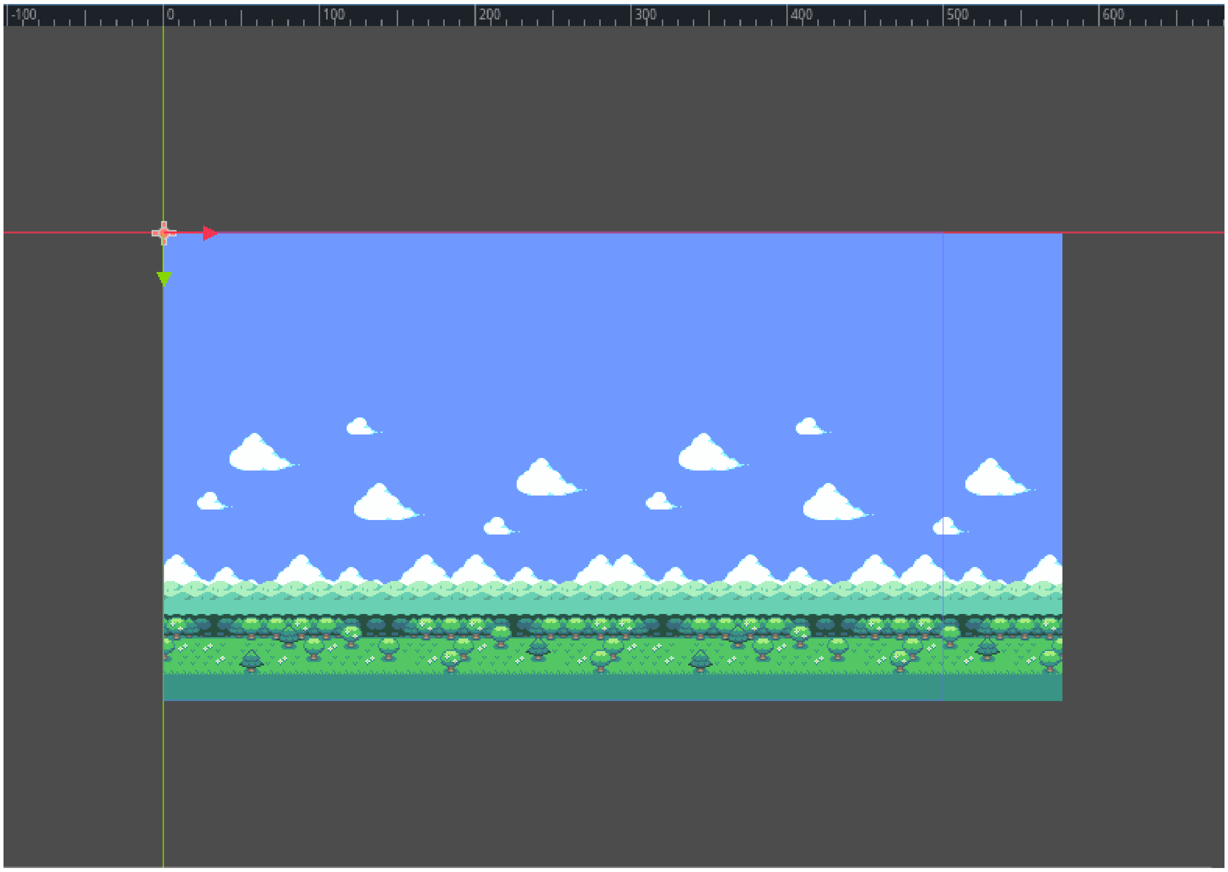


Repeat the textures

You can also start off on the right foot by preparing child nodes earlier in the process. If you have a Sprite2D you'd like to repeat, but is too small, you can do the following to repeat it:

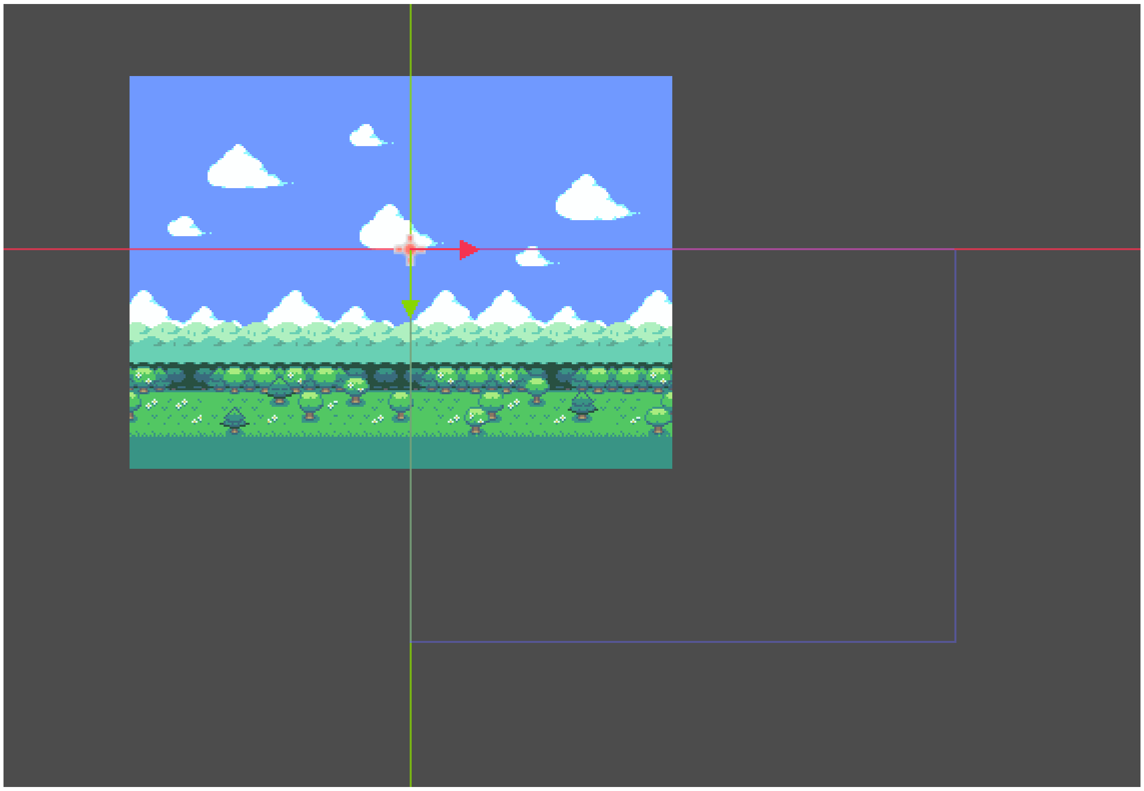
- set texture_repeat to CanvasItem.TEXTURE_REPEAT_ENABLED
- set region_enabled to `true`
- set the region_rect to a multiple of the size of your texture large enough to cover the viewport.

Below, you can see that repeating the image twice makes it large enough to cover the screen.

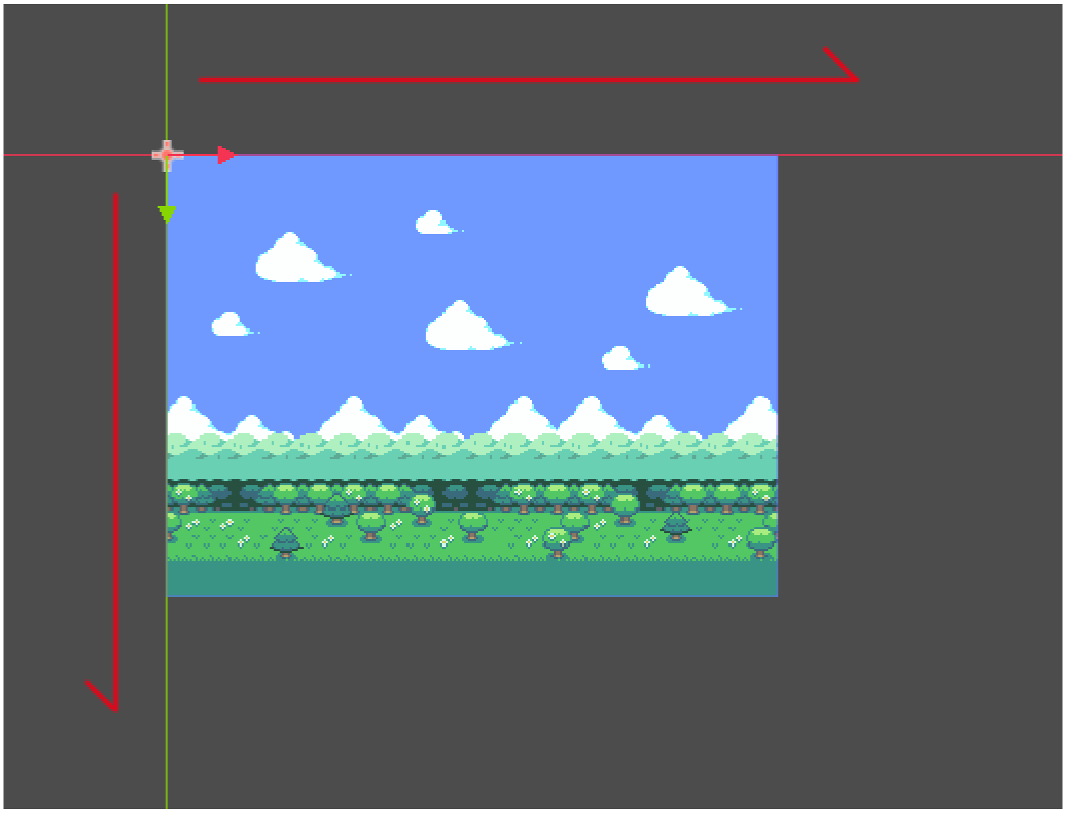

-100
0
100
200
300
400
500
600

Poor positioning

It's common to see users mistakenly set all of their textures to be centered at `(0,0)`:



This creates problems with the infinite repeat effect and should be avoided. The "infinite repeat canvas" starts at `(0,0)` and expands down and to the right to the size of the repeat_size value.

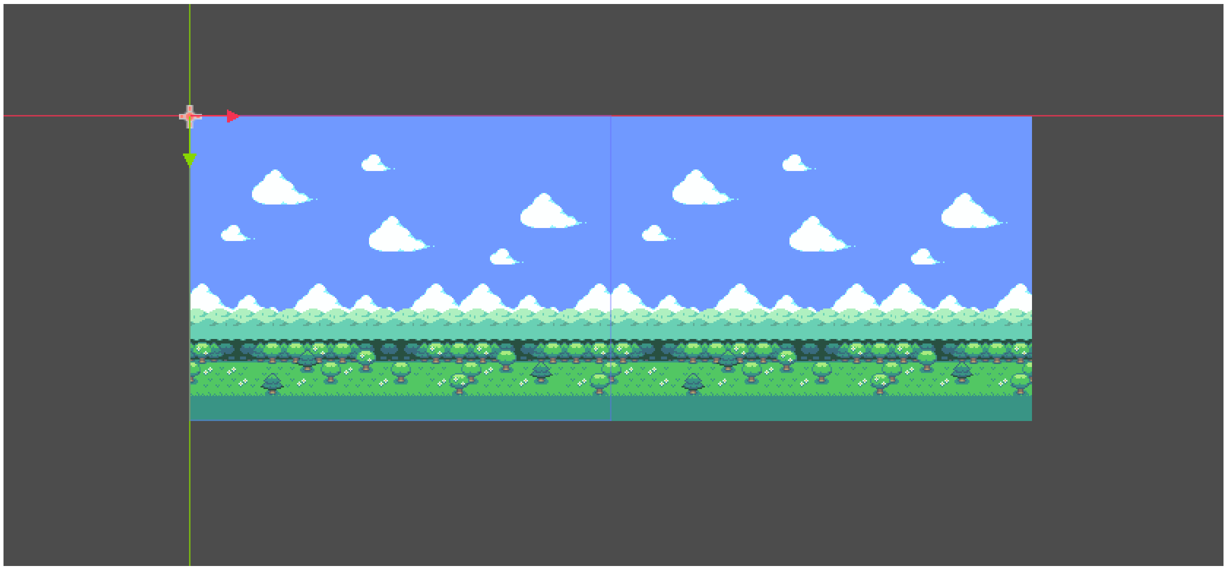


If the textures are centered on the `(0,0)` crossing, the infinite repeat canvas is only partly covered, so it only partly repeats.

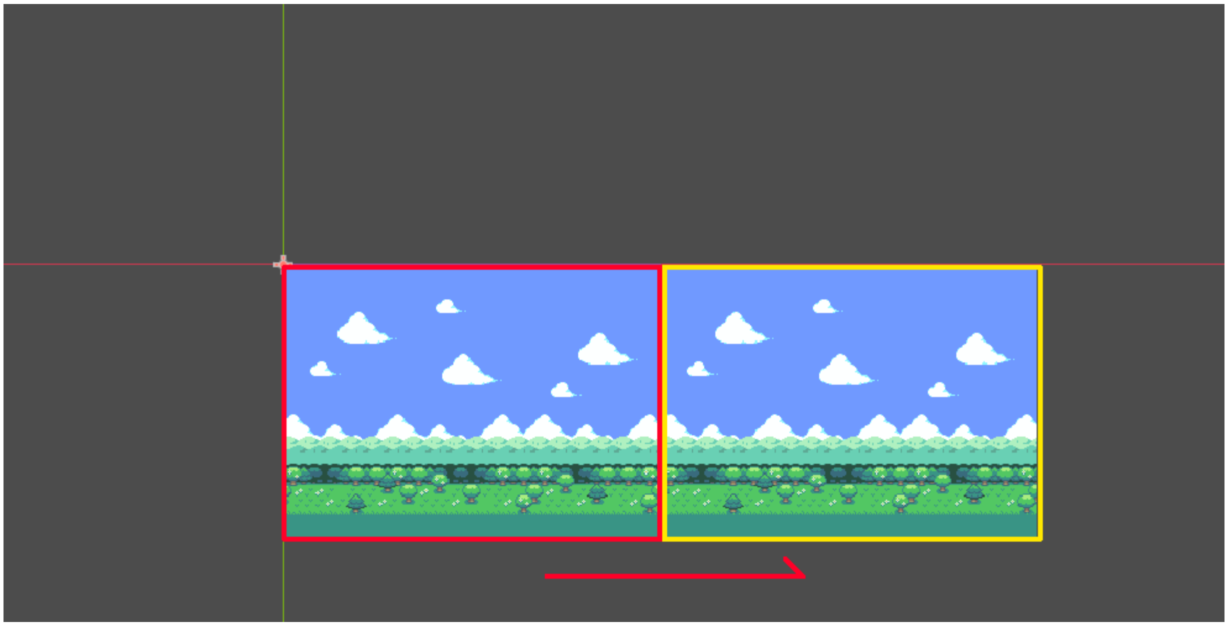
Would increasing `repeat_times` fix this?

Increasing repeat_times technically *would* work in some scenarios, but is a brute force solution and not the problem it is designed to solve (we'll go over this in a bit). A better fix is to understand how the repeat effect works and set up the parallax textures appropriately to begin with.

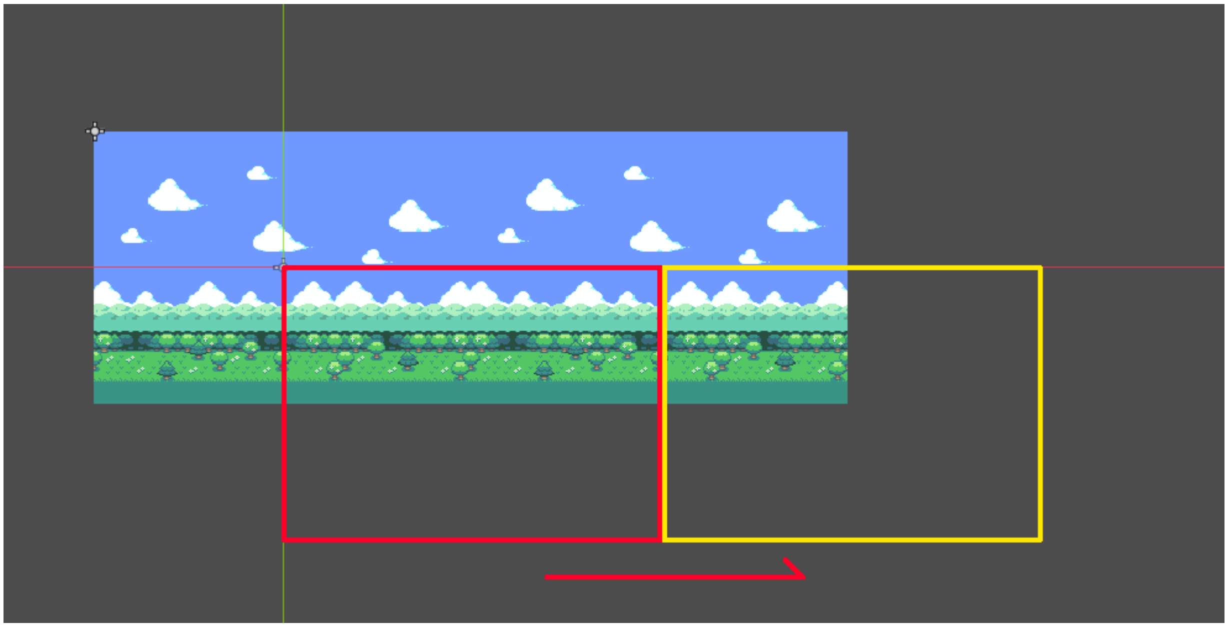
First, check to see if any textures are spilling over onto the negative parts of the canvas. Make sure the textures used in the parallax nodes fit inside the "infinite repeat canvas" starting at `(0,0)`. That way, if Parallax2D.repeat_size is set correctly, it should look something like this, with one single loop of the image the same size or larger than the viewport:



If you think of how the image scrolls across the screen, it starts by displaying what's inside the red rectangle (determined by [repeat_size](#)), and when it reaches what's inside the yellow rectangle it zips the image forward to give the illusion of scrolling forever.



If you have the image positioned away from the "infinite repeat canvas", when the camera reaches the yellow rectangle, half of the image is cut off before it jumps forward like in the image below:

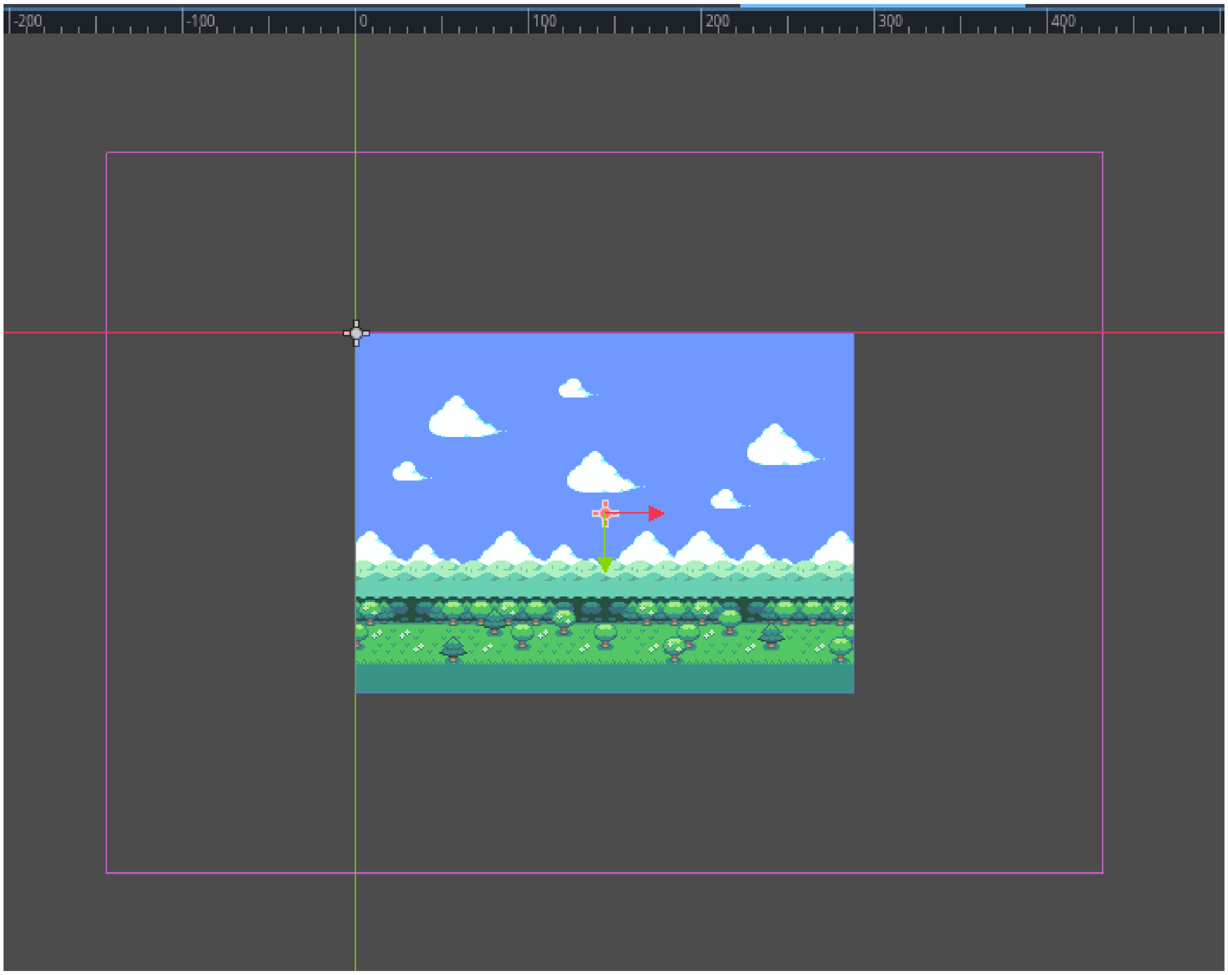


Scroll offset

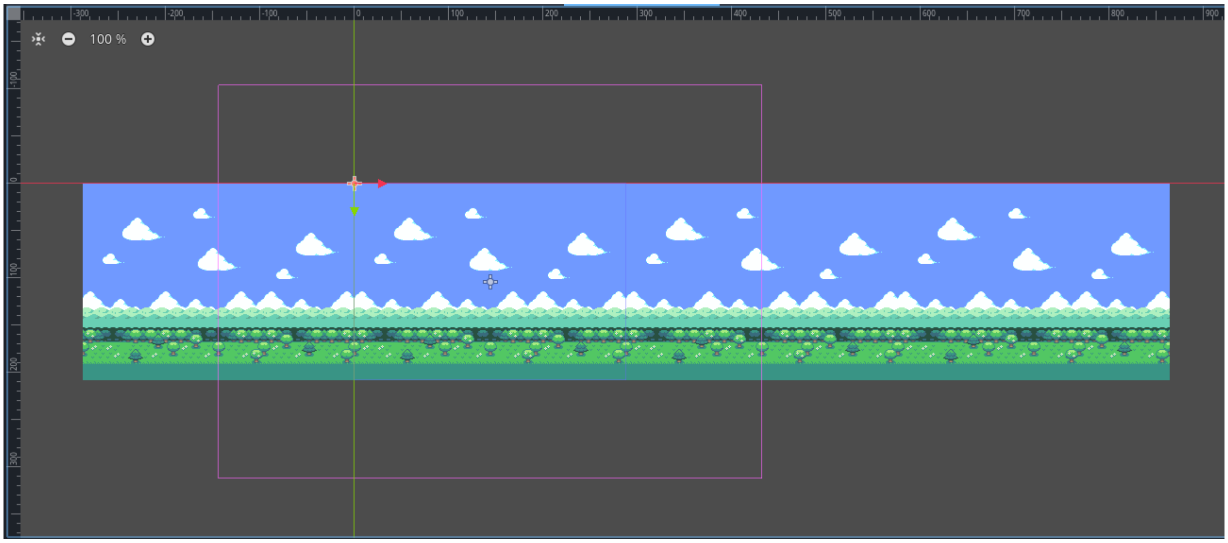
If your parallax textures are already working correctly, but you prefer it to start at a different point, Parallax2D comes with a scroll_offset property used to offset where the infinite repeat canvas starts. As an example, if your image is 288x208, setting the scroll_offset to `(-144,0)` or `(144,0)` allows it to begin halfway across the image.

Repeat times

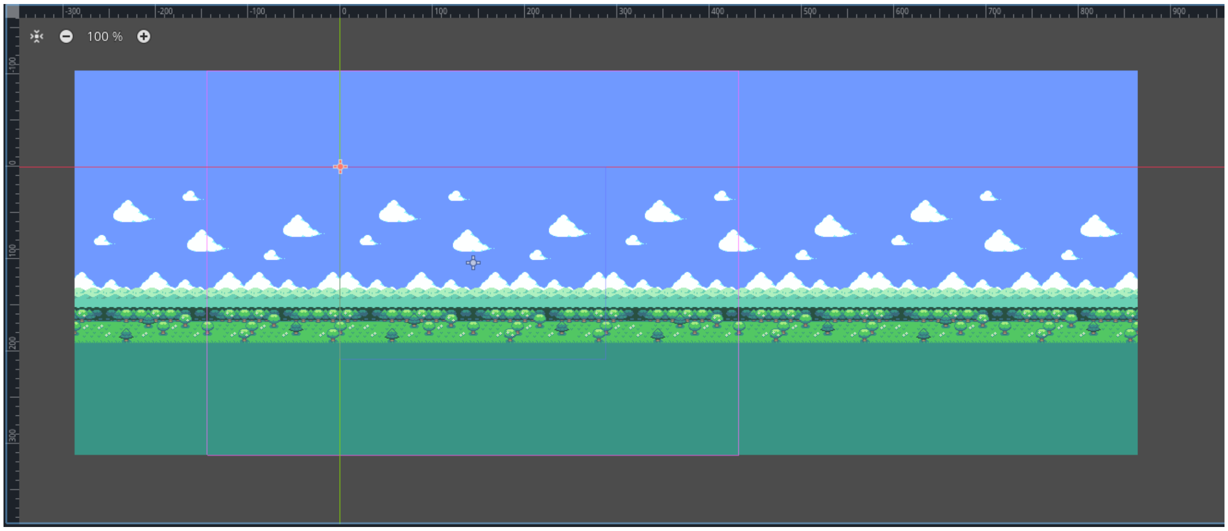
Ideally, following this guide, your parallax textures are large enough to cover the screen even when zoomed out. Until now, we have had a perfectly fitting 288x208 texture inside of a 288x208 viewport. However, problems occur when we zoom out by setting the Camera2D.zoom to `(0.5, 0.5)`:



Even though everything is correctly set for the viewport at the default zoom level, zooming out makes it smaller than the viewport, breaking the infinite repeat effect. This is where [repeat_times](#) can help out. Setting a value of `3` (one extra repeat behind and in front), it is now large enough to accommodate the infinite repeat effect.



If these textures were meant to be repeated vertically, we would have specified a `y` value for the [repeat_size](). The [repeat_times]() would automatically add a repeat above and below as well. This is only a horizontal parallax, so it leaves an empty block above and below the image. How do we solve this? We need to get creative! In this example, we stretch the sky higher, and grass sprite lower. The textures now support the normal zoom level and zooming out to half size.

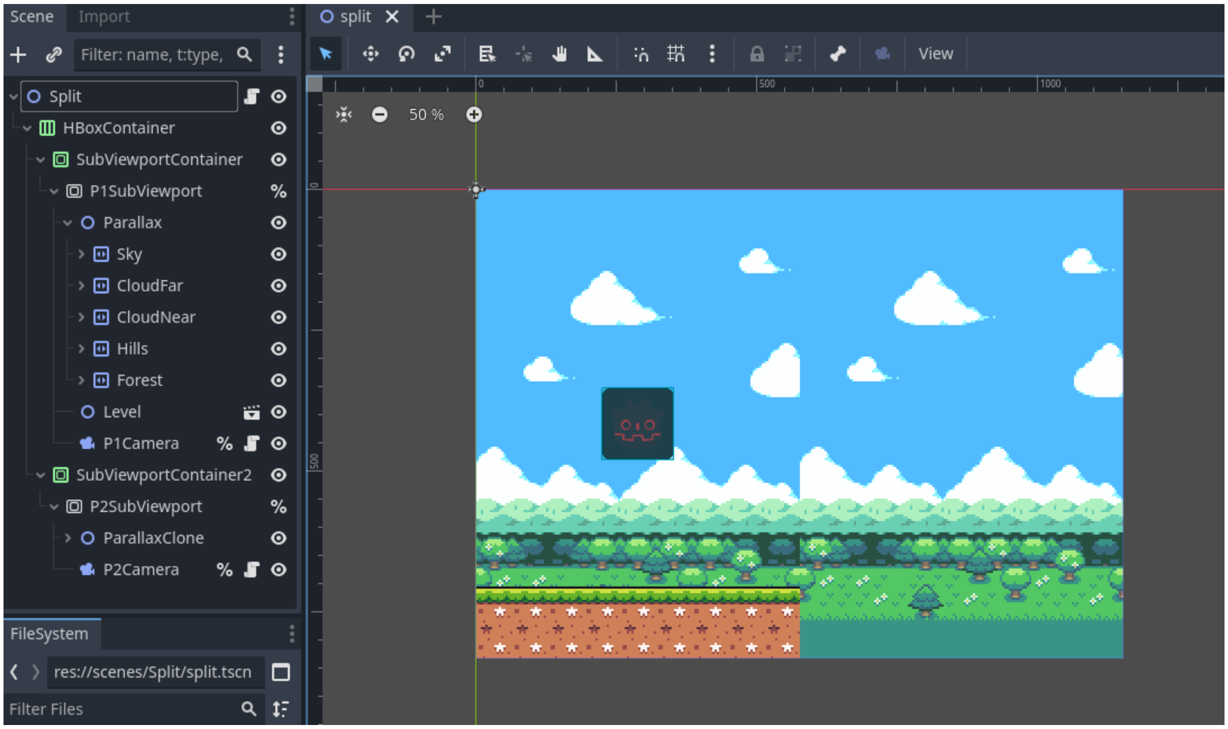


Split screen

Most tutorials for making a split screen game in Godot begin by writing a small script to assign the Viewport.world_2d of the first SubViewport to the second, so they have a shared display. Questions often pop up about how to share a parallax effect between both screens.

The parallax effect fakes a perspective by moving the positions of different textures in relation to the camera. This is understandably problematic if you have multiple cameras, because your textures can't be in two places at once!

This is still achievable by cloning the parallax nodes into the second (or third or fourth) SubViewport. Here's how a setup looks for a two player game:


Scene
Import
Filter: name, t:type,
Split
HBoxContainer
SubViewportContainer
P1SubViewport
Parallax
Sky
CloudFar
CloudNear
Hills
Forest
Level
P1Camera
SubViewportContainer2
P2SubViewport
ParallaxClone
P2Camera
FileSystem
res://scenes/Split/split.tscn
Filter Files
split
View
50 %
0
500
1000
500

Of course, now both backgrounds show in both SubViewports. What we want is for each parallax to only show in their corresponding viewport. We can achieve this by doing the following:

- Leave all parallax nodes at their default visibility_layer of 1.

- Set the first SubViewport's canvas_cull_mask to only layers 1 and 2.
- Do the same for the second SubViewport but use layers 1 and 3.
- Give your parallax nodes in the first SubViewport a common parent and set its visibility_layer to 2.
- Do the same for the second SubViewport's parallax nodes, but use a layer of 3.

How does this work? If a canvas item has a visibility_layer that doesn't match the SubViewport's canvas_cull_mask, it will hide all children, even if they do. We use this to our advantage, letting the SubViewports cut off rendering of parallax nodes whose parent doesn't have a supported visibility_layer.

Previewing in the editor

Prior to 4.3, the recommendation was to place every layer in their own ParallaxBackground, enable the follow_viewport_enabled property, and scale the individual layer. This method has always been tricky to get right, but is still achievable by using a CanvasLayer instead of a ParallaxBackground.

Note

Another recommendation is KoBeWi's "Parallax2D Preview" addon [https://github.com/KoBeWi/Godot-Parallax2D-Preview]. It provides a few different preview modes and is very handy!

2D movement overview

Introduction

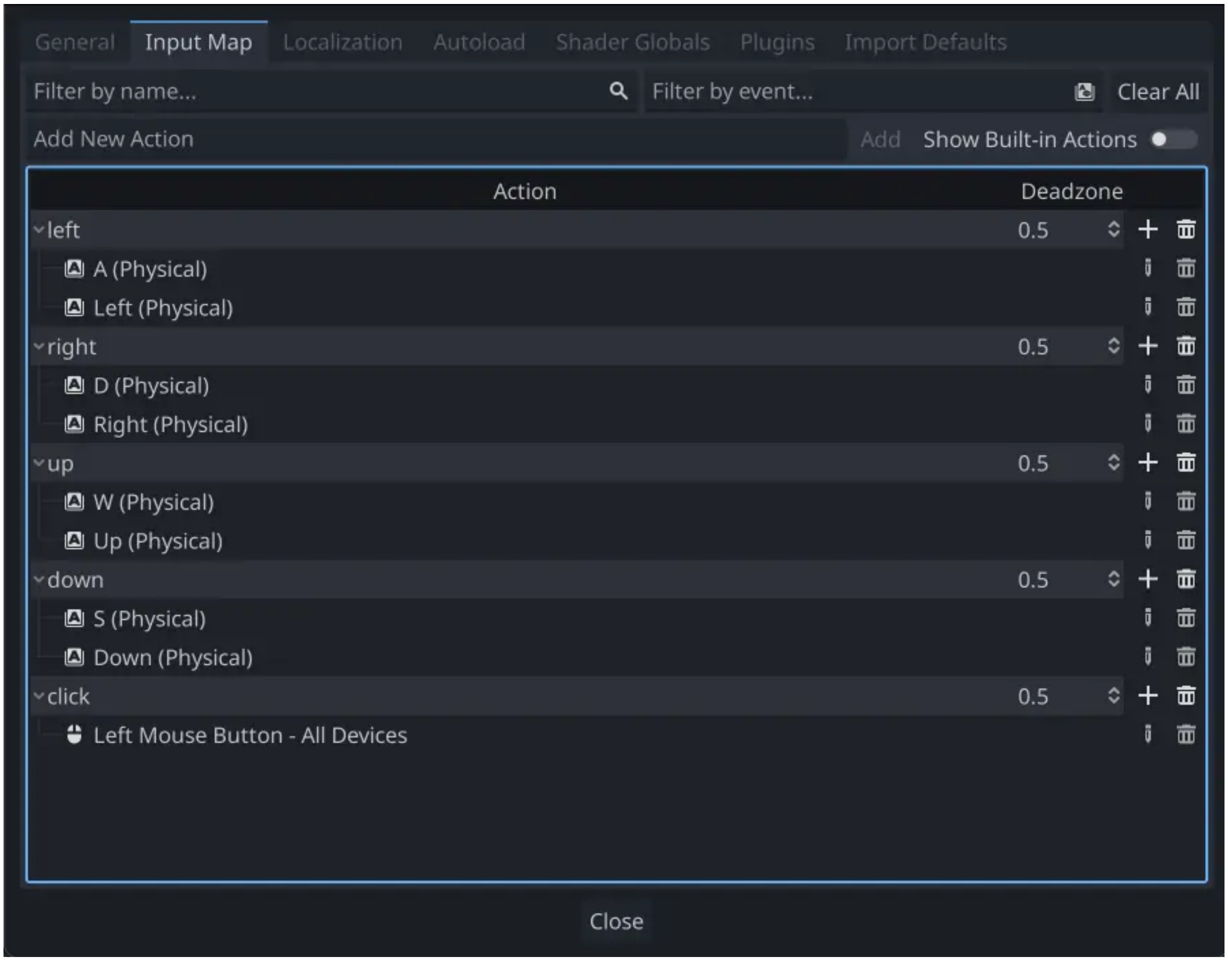
Every beginner has been there: "How do I move my character?" Depending on the style of game you're making, you may have special requirements, but in general the movement in most 2D games is based on a small number of designs.

We'll use CharacterBody2D for these examples, but the principles will apply to other node types (Area2D, RigidBody2D) as well.

Setup

Each example below uses the same scene setup. Start with a `CharacterBody2D` with two children: `Sprite2D` and `CollisionShape2D`. You can use the Godot icon ("icon.png") for the Sprite2D's texture or use any other 2D image you have.

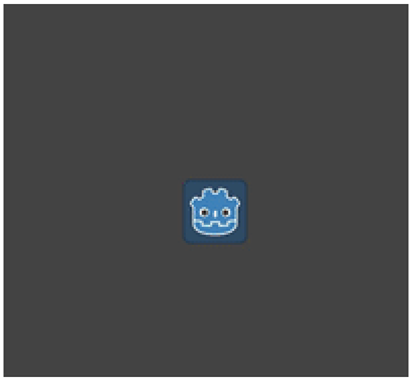
Open `Project -> Project Settings` and select the "Input Map" tab. Add the following input actions (see InputEvent for details):


General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Filter by name...
Filter by event...
Clear All
Add New Action
Add
Show Built-in Actions
Close

Action	Deadzone
left	0.5
A (Physical)	
Left (Physical)	
right	0.5
D (Physical)	
Right (Physical)	
up	0.5
W (Physical)	
Up (Physical)	
down	0.5
S (Physical)	
Down (Physical)	
click	0.5
Left Mouse Button - All Devices	

8-way movement

In this scenario, you want the user to press the four directional keys (up/left/down/right or W/A/S/D) and move in the selected direction. The name "8-way movement" comes from the fact that the player can move diagonally by pressing two keys at the same time.



Add a script to the character body and add the following code:

GDScript

```
extends CharacterBody2D

@export var speed = 400

func get_input():
    var input_direction = Input.get_vector("left", "right", "up", "down")
    velocity = input_direction * speed

func _physics_process(delta):
    get_input()
    move_and_slide()
```

C#

```
using Godot;

public partial class Movement : CharacterBody2D
{
    [Export]
    public int Speed { get; set; } = 400;

    public void GetInput()
    {

        Vector2 inputDirection = Input.GetVector("left", "right", "up", "down");
        Velocity = inputDirection * Speed;
    }

    public override void _PhysicsProcess(double delta)
    {
        GetInput();
        MoveAndSlide();
    }
}
```

In the `get_input()` function, we use Input `get_vector()` to check for the four key events and sum return a direction vector.

We can then set our velocity by multiplying this direction vector, which has a length of `1`, by our desired speed.

Tip

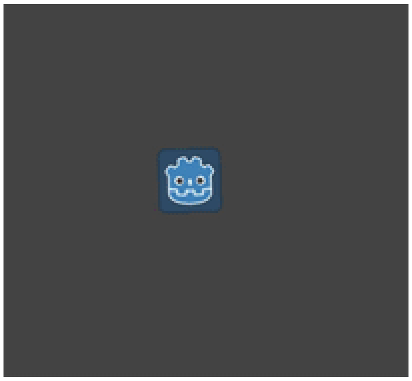
If you've never used vector math before, or need a refresher, you can see an explanation of vector usage in Godot at Vector math.

Note

If the code above does nothing when you press the keys, double-check that you've set up input actions correctly as described in the Setup part of this tutorial.

Rotation + movement

This type of movement is sometimes called "Asteroids-style" because it resembles how that classic arcade game worked. Pressing left/right rotates the character, while up/down moves it forward or backward in whatever direction it's facing.



GDScript

```
extends CharacterBody2D

@export var speed = 400
@export var rotation_speed = 1.5

var rotation_direction = 0

func get_input():
    rotation_direction = Input.get_axis("left", "right")
    velocity = transform.x * Input.get_axis("down", "up") *
speed

func _physics_process(delta):
    get_input()
    rotation += rotation_direction * rotation_speed * delta
    move_and_slide()
```

C#

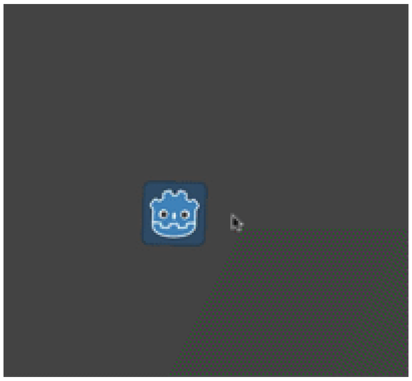
```
using Godot;

public partial class Movement : CharacterBody2D
{
    [Export]
    public int Speed { get; set; } = 400;

    [Export]
    public float RotationSpeed { get; set; } = 1.5f;

    private float _rotationDirection;
    public void GetInput()
    {
        _rotationDirection = Input.GetAxis("left", "right");
        Velocity = Transform.X * Input.GetAxis("down", "up") *
Speed;
    }

    public override void _PhysicsProcess(double delta)
    {
        GetInput();
        Rotation += _rotationDirection * RotationSpeed *
(float)delta;
        MoveAndSlide();
    }
}
```

Here we've added two variables to track our rotation direction and speed. The rotation is applied directly to the body's `rotation` property.

To set the velocity, we use the body's `transform.x` which is a vector pointing in the body's "forward" direction, and multiply that by the speed.

Rotation + movement (mouse)

This style of movement is a variation of the previous one. This time, the direction is set by the mouse position instead of the keyboard. The character will always "look at" the mouse pointer. The forward/back inputs remain the same, however.



GDScript

```
extends CharacterBody2D

@export var speed = 400

func get_input():
    look_at(get_global_mouse_position())
    velocity = transform.x * Input.get_axis("down", "up") *
speed

func _physics_process(delta):
    get_input()
    move_and_slide()
```

C#

```
using Godot;

public partial class Movement : CharacterBody2D
{
    [Export]
    public int Speed { get; set; } = 400;

    public void GetInput()
    {
        LookAt(GetGlobalMousePosition());
        Velocity = Transform.X * Input.GetAxis("down", "up") *
Speed;
    }

    public override void _PhysicsProcess(double delta)
    {
        GetInput();
        MoveAndSlide();
    }
}
```

Here we're using the Node2D `look_at()` method to point the player towards the mouse's position. Without this function, you could get the same effect by setting the angle like this:

GDScript

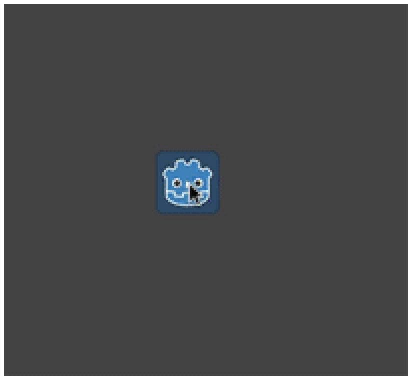
```
rotation = get_global_mouse_position().angle_to_point(position)
```

C#

```
var rotation = GetGlobalMousePosition().AngleToPoint(Position);
```

Click-and-move

This last example uses only the mouse to control the character. Clicking on the screen will cause the player to move to the target location.



GDScript

```
extends CharacterBody2D
```

```
@export var speed = 400

var target = position

func _input(event):
    # Use is_action_pressed to only accept single taps as input instead of mouse drags.
    if event.is_action_pressed(&"click"):
        target = get_global_mouse_position()

func _physics_process(delta):
    velocity = position.direction_to(target) * speed
    # look_at(target)
    if position.distance_to(target) > 10:
        move_and_slide()
```

C#

```
using Godot;

public partial class Movement : CharacterBody2D
{
    [Export]
    public int Speed { get; set; } = 400;

    private Vector2 _target;

    public override void _Input(InputEvent @event)
    {
        // Use IsActionPressed to only accept single taps as input instead of mouse drags.
        if (@event.IsActionPressed("click"))
        {
            _target = GetGlobalMousePosition();
        }
    }

    public override void _PhysicsProcess(double delta)
    {
        Velocity = Position.DirectionTo(_target) * Speed;
        // LookAt(_target);
        if (Position.DistanceTo(_target) > 10)
        {
            MoveAndSlide();
        }
    }
}
```

Note the `distance_to()` check we make prior to movement. Without this test, the body would "jitter" upon reaching the target position, as it moves slightly past the position and tries to move back, only to move too far and repeat.

Uncommenting the `look_at()` line will also turn the body to point in its direction of motion if you prefer.

Tip

This technique can also be used as the basis of a "following" character. The `target` position can be that of any object you want to move to.

Summary

You may find these code samples useful as starting points for your own projects. Feel free to use them and experiment with them to see what you can make.

You can download this sample project here: [2d_movement_starter.zip](https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/2d_movement_starter.zip) [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/2d_movement_starter.zip]

Using TileSets

Introduction

A tilemap is a grid of tiles used to create a game's layout. There are several benefits to using TileMapLayer nodes to design your levels. First, they let you draw a layout by "painting" tiles onto a grid, which is much faster than placing individual Sprite2D nodes one by one. Second, they allow for larger levels because they are optimized for drawing large numbers of tiles. Finally, they allow you to add greater functionality to your tiles with collision, occlusion, and navigation shapes.

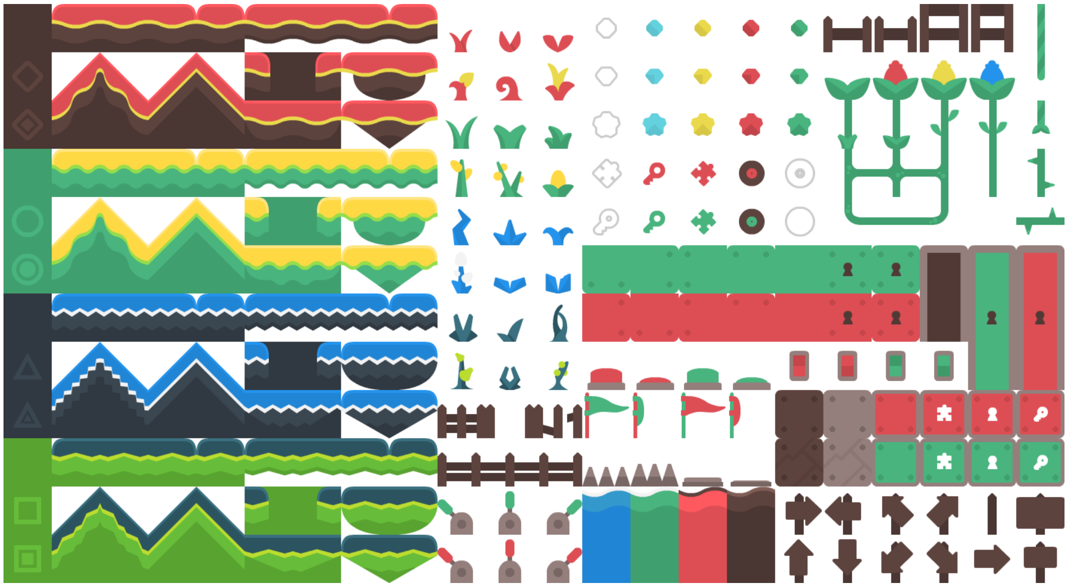
To use TileMapLayer nodes, you will need to create a TileSet first. A TileSet is a collection of tiles that can be placed in a TileMapLayer node. After creating a TileSet, you will be able to place them using the TileMap editor.

To follow this guide, you will need an image containing your tiles where every tile has the same size (large objects can be split into several tiles). This image is called a *tilesheet*. Tiles do not have to be square: they can be rectangular, hexagonal, or isometric (pseudo-3D perspective).

Creating a new TileSet

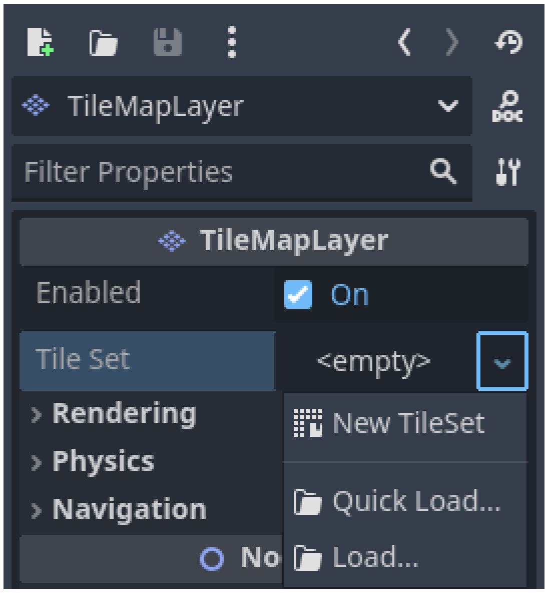
Using a tilesheet

This demonstration will use the following tiles taken from Kenney's "Abstract Platformer" pack [https://kenney.nl/assets/abstract-platformer]. We'll use this particular *tilesheet* from the set:



Tilesheet with 64×64 tiles. Credit: Kenney [https://kenney.nl/assets/abstract-platformer]

Create a new **TileMapLayer** node, then select it and create a new TileSet resource in the inspector:

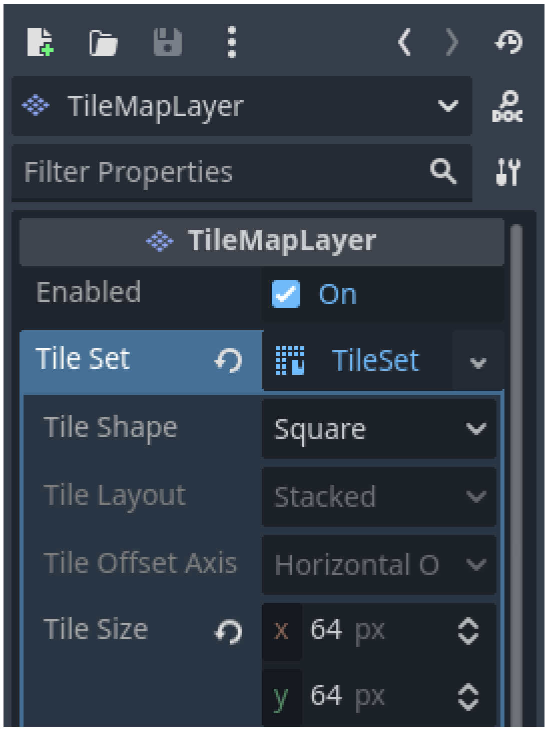


TileMapLayer
Filter Properties
TileMapLayer
Enabled
On
Tile Set
<empty>
Rendering
Physics
Navigation
No
New TileSet
Quick Load...
Load...

Creating a new TileSet resource within the TileMapLayer node

After creating the TileSet resource, click the value to unfold it in the inspector. The default tile shape is Square, but you can also choose Isometric, Half-Offset Square or Hexagon (depending on the shape of your tile images). If using a tile shape other than Square, you may also need to adjust the **Tile Layout** and **Tile Offset Axis** properties. Lastly, enabling the **Rendering > UV Clipping** property may be useful if you wish tiles to be clipped by their tile coordinates. This ensures tiles cannot draw outside their allocated area on the tilesheet.

Set the tile size to 64×64 in the inspector to match the example tilesheet:



Enabled ☑ On

Tile Set: TileSet

Tile Shape: Square

Tile Layout: Stacked

Tile Offset Axis: Horizontal O

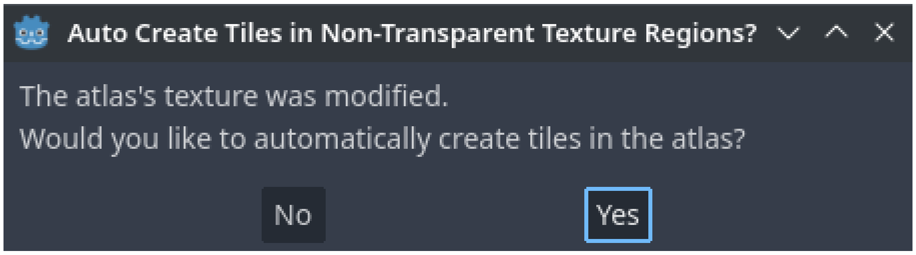
Tile Size: x 64 px, y 64 px

Setting the tile size to 64×64 to match the example tilesheet

If relying on automatic tiles creation (like we're about to do here), you must set the tile size **before** creating the *atlas*. The atlas will determine which tiles from the tilesheet can be added to a

TileMapLayer node (as not every part of the image may be a valid tile).

Open the **TileSet** panel at the bottom of the editor, then click and drag the tilesheet image onto the panel. You will be asked whether to create tiles automatically. Answer **Yes**:



Auto Create Tiles in Non-Transparent Texture Regions?

The atlas's texture was modified.
Would you like to automatically create tiles in the atlas?

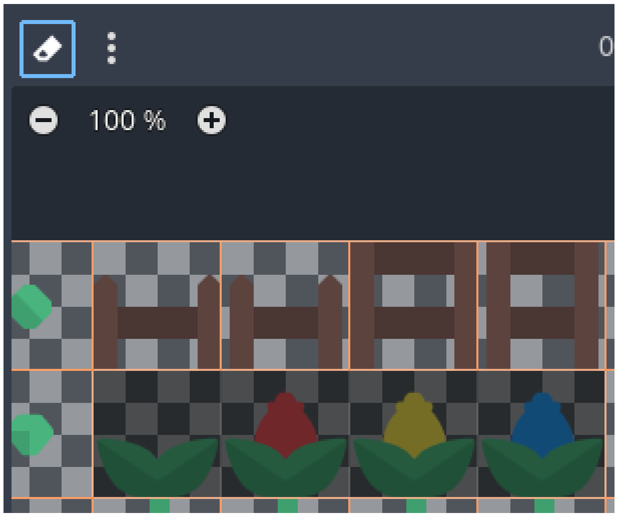
Automatically creating tiles based on tilesheet image content

This will automatically create tiles according to the tile size you specified earlier in the TileSet resource. This greatly speeds up initial tile setup.

Note

When using automatic tile generation based on image contents, parts of the tilesheet that are *fully* transparent will not have tiles generated.

If there are tiles from the tilesheet you do not wish to be present in atlas, choose the Eraser tool at the top of the tileset preview, then click the tiles you wish to remove:



Using the Eraser tool to remove unwanted tiles from the TileSet atlas

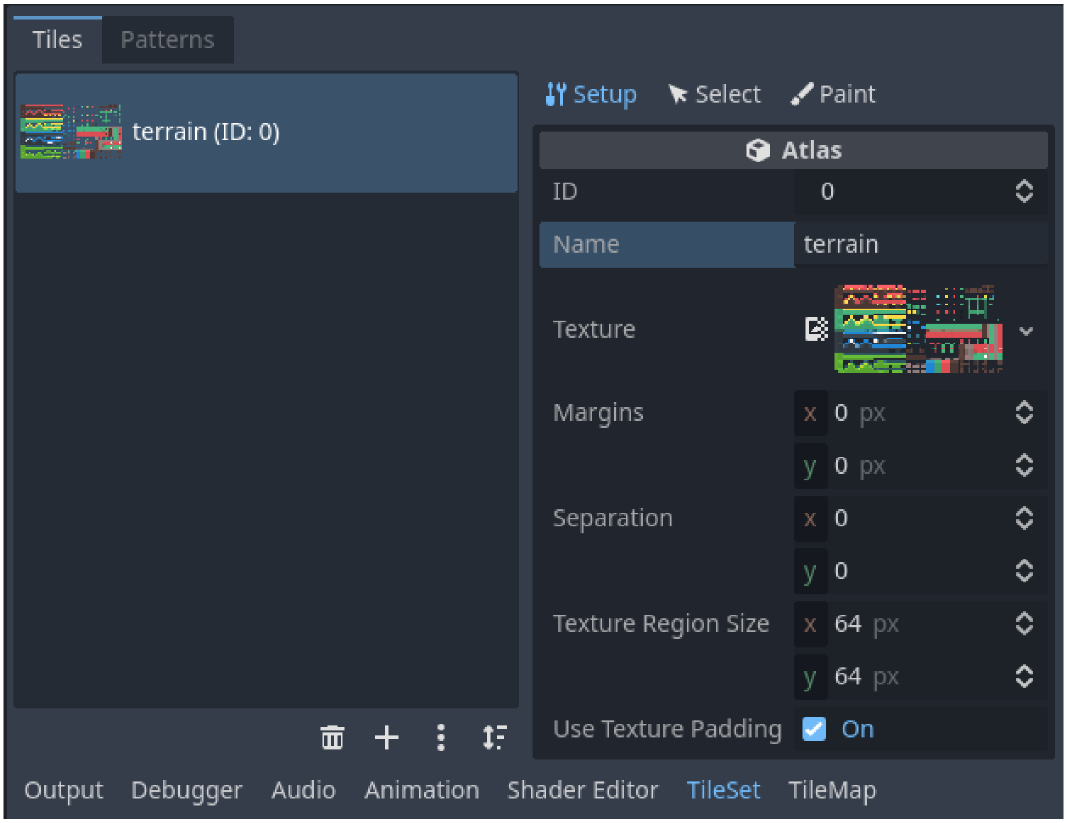
You can also right-click a tile and choose **Delete**, as an alternative to the Eraser tool.

Tip

Like in the 2D and TileMap editors, you can pan across the TileSet panel using the middle or right mouse buttons, and zoom using the mouse wheel or buttons in the top-left corner.

If you wish to source tiles from several tilesheet images for a single TileSet, create additional atlases and assign textures to each of them before continuing. It is also possible to use one image per tile this way (although using tilesheets is recommended for better usability).

You can adjust properties for the atlas in the middle column:


Tiles
Patterns
terrain (ID: 0)
Setup
Select
Paint
Atlas
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

ID: 0
Name: terrain
Texture
Margins: x 0 px, y 0 px
Separation: x 0, y 0
Texture Region Size: x 64 px, y 64 px
Use Texture Padding: ☑ On

Adjusting TileSet atlas properties in the dedicated inspector (part of the TileSet panel)

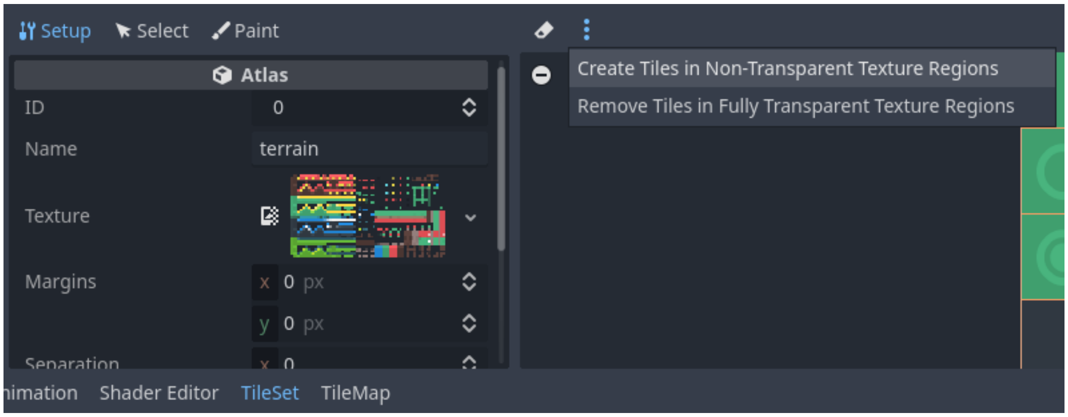
The following properties can be adjusted on the atlas:

- **ID:** The identifier (unique within this TileSet), used for sorting.
- **Name:** The human-readable name for the atlas. Use a descriptive name here for organizational purposes (such as "terrain", "decoration", etc).
- **Margins:** The margins on the image's edges that should not be selectable as tiles (in pixels). Increasing this can be useful if you download a tilesheet image that has margins on the edges (e.g. for attribution).
- **Separation:** The separation between each tile on the atlas in pixels. Increasing this can be useful if the tilesheet image you're using contains guides (such as outlines between every tile).
- **Texture Region Size:** The size of each tile on the atlas in pixels. In most cases, this should match the tile size defined in

the TileMapLayer property (although this is not strictly necessary).

- **Use Texture Padding:** If checked, adds a 1-pixel transparent edge around each tile to prevent texture bleeding when filtering is enabled. It's recommended to leave this enabled unless you're running into rendering issues due to texture padding.

Note that changing texture margin, separation and region size may cause tiles to be lost (as some of them would be located outside the atlas image's coordinates). To regenerate tiles automatically from the tilesheet, use the three vertical dots menu button at the top of the TileSet editor and choose **Create Tiles in Non-Transparent Texture Regions**:


Setup
Select
Paint
Atlas
Create Tiles in Non-Transparent Texture Regions
Remove Tiles in Fully Transparent Texture Regions
Texture
Separation
imation
Shader Editor
TileSet
TileMap

ID 0

Name terrain

Margins x 0 px

y 0 px

Recreating tiles automatically after changing atlas properties

Using a collection of scenes

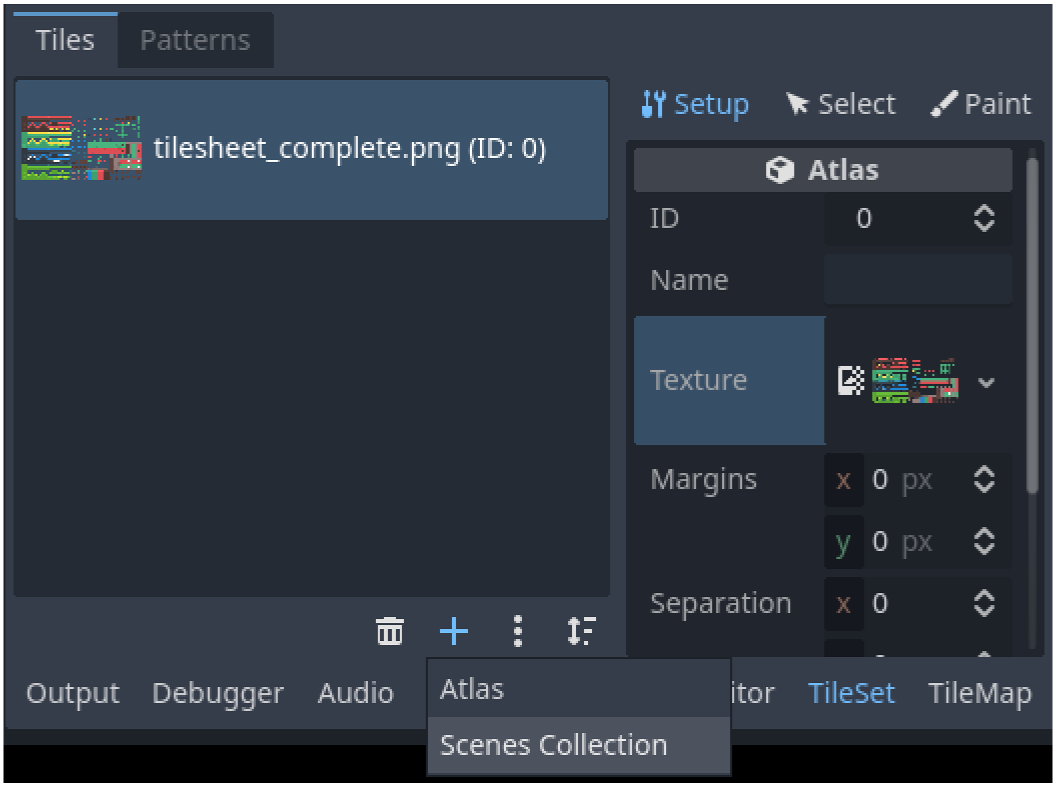
Since Godot 4.0, you can place actual *scenes* as tiles. This allows you to use any collection of nodes as a tile. For example, you could use scene tiles to place gameplay elements, such as shops the player may be able to interact with. You could also use scene tiles to place AudioStreamPlayer2Ds (for ambient sounds), particle effects, and more.

Warning

Scene tiles come with a greater performance overhead compared to atlases, as every scene is instanced individually for every placed tile.

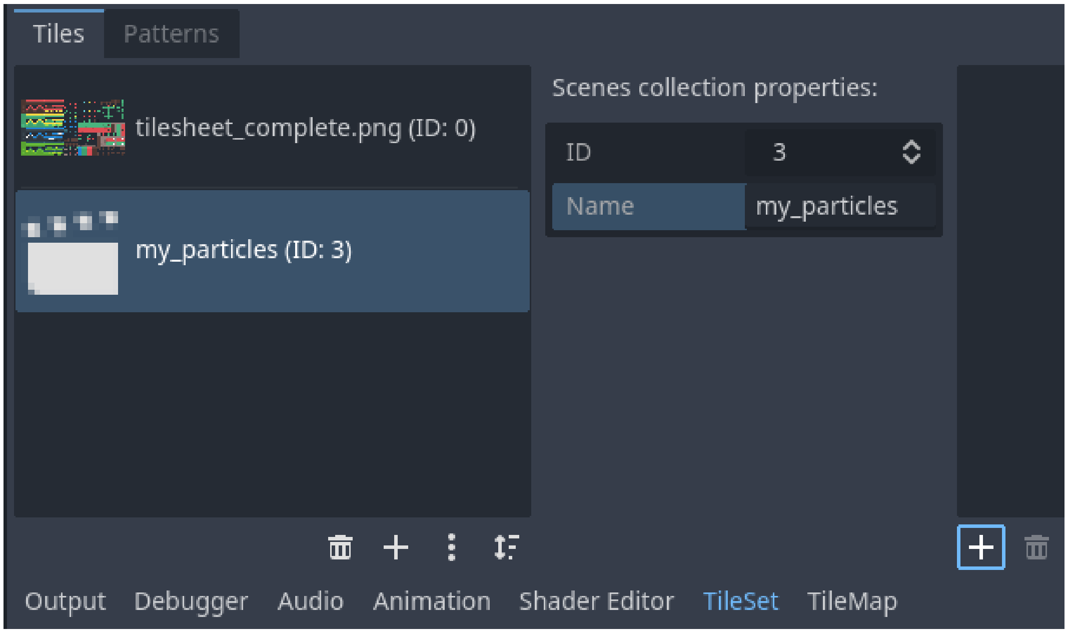
It's recommended to use only scene tiles when necessary. To draw sprites in a tile without any kind of advanced manipulation, use atlases instead.

For this example, we'll create a scene containing a CPUParticles2D root node. Save this scene to a scene file (separate from the scene containing the TileMapLayer), then switch to the scene containing the TileMapLayer node. Open the TileSet editor, and create a new **Scenes Collection** in the left column:


Tiles
Patterns
tilesheet_complete.png (ID: 0)
Setup
Select
Paint
Atlas
ID
0
Name
Texture
Margins
x 0 px
y 0 px
Separation
x 0
Output
Debugger
Audio
Atlas
Scenes Collection
itor
TileSet
TileMap

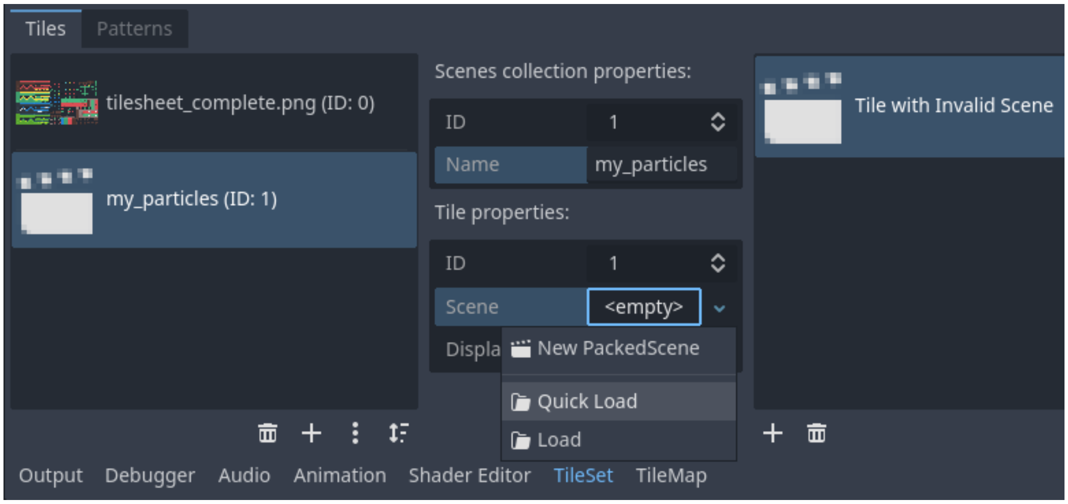
Creating a scenes collection in the TileSet editor

After creating a scenes collection, you can enter a descriptive name for the scenes collection in the middle column if you wish. Select this scenes collection then create a new scene slot:


Tiles
Patterns
tilesheet_complete.png (ID: 0)
my_particles (ID: 3)
Scenes collection properties:
ID
3
Name
my_particles
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Creating a scene tile after selecting the scenes collection in the TileSet editor

Select this scene slot in the right column, then use **Quick Load** (or **Load**) to load the scene file containing the particles:


Tiles
Patterns
tilesheet_complete.png (ID: 0)
my_particles (ID: 1)
Scenes collection properties:
ID
1
Name
my_particles
Tile properties:
ID
1
Scene
<empty>
Displa
New PackedScene
Quick Load
Load
Tile with Invalid Scene
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

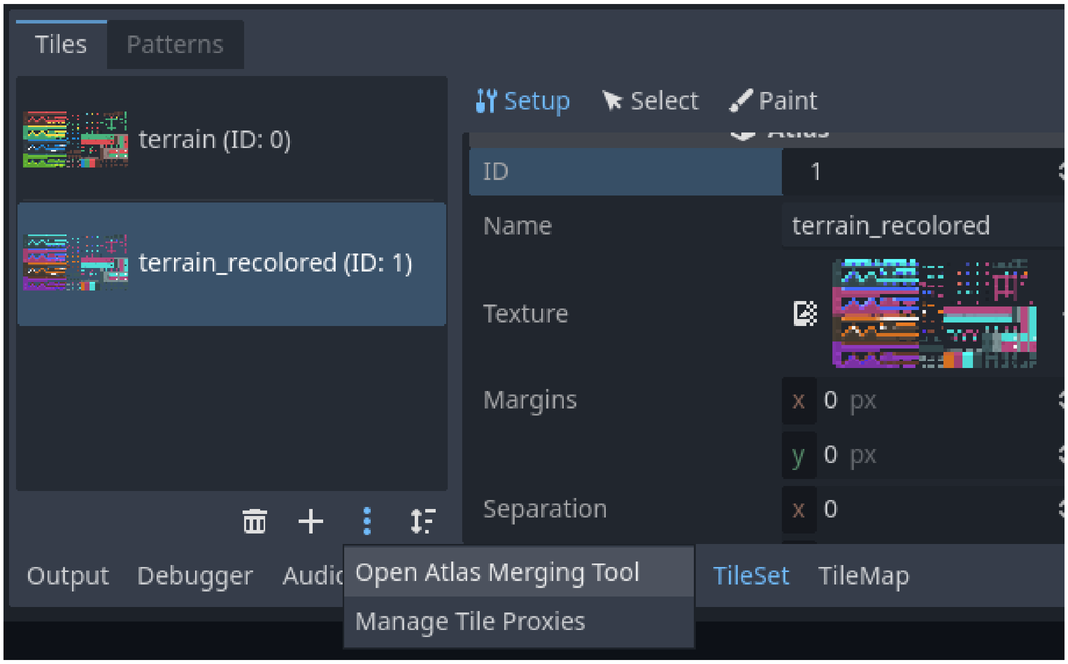
Creating a scene slot, then loading a scene file into it in the TileSet editor

You now have a scene tile in your TileSet. Once you switch to the TileMap editor, you'll be able to select it from the scenes collection and paint it like any other tile.

Merging several atlases into a single atlas

Using multiple atlases within a single TileSet resource can sometimes be useful, but it can also be cumbersome in certain situations (especially if you're using one image per tile). Godot allows you to merge several atlases into a single atlas for easier organization.

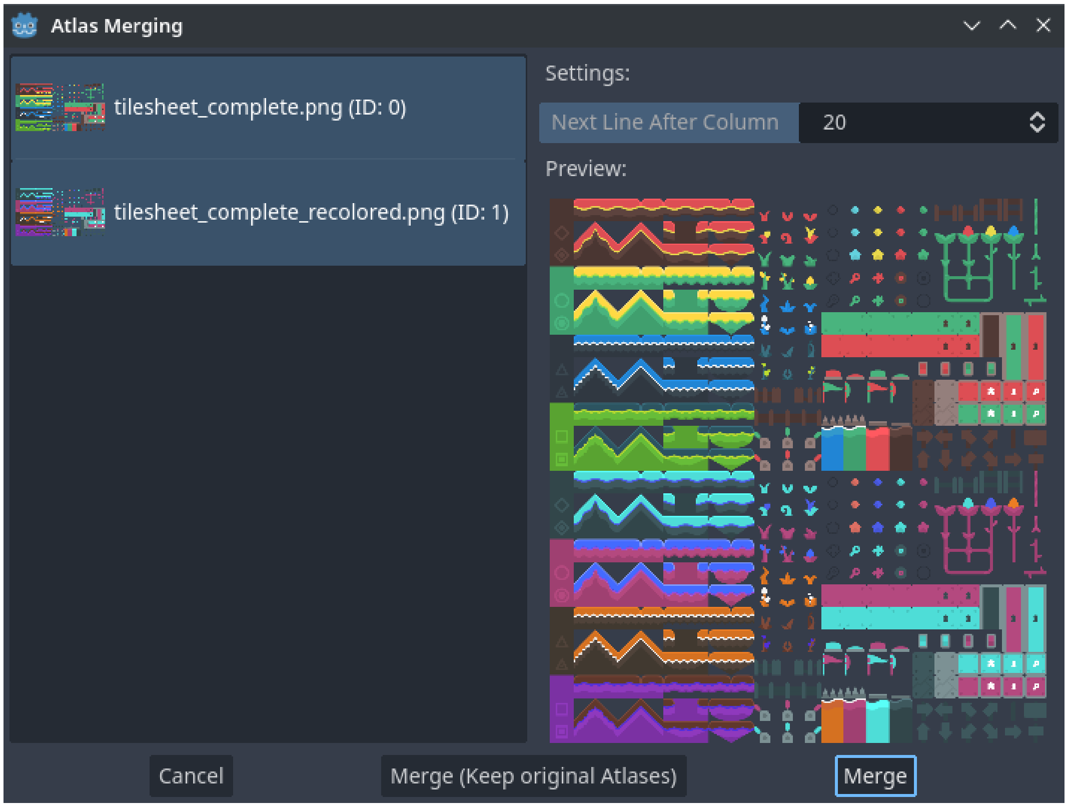
To do so, you must have more than one atlas created in the TileSet resource. Use the "three vertical dots" menu button located at the bottom of the list of atlases, then choose **Open Atlas Merging Tool**:


Tiles
Patterns
terrain (ID: 0)
terrain_recolored (ID: 1)
Setup
Select
Paint
Open Atlas Merging Tool
Manage Tile Proxies
Output
Debugger
Audio
TileSet
TileMap

ID: 1
Name: terrain_recolored
Texture
Margins: x 0 px, y 0 px
Separation: x 0

Opening the atlas merging tool after creating multiple atlases

This will open a dialog, in which you can select several atlases by holding `Shift` or `Ctrl` then clicking on multiple elements:


Atlas Merging
tilesheet_complete.png (ID: 0)
tilesheet_complete_recolored.png (ID: 1)
Settings:
Next Line After Column
20
Preview:
Cancel
Merge (Keep original Atlases)
Merge

Using the atlas merging tool dialog

Choose **Merge** to merge the selected atlases into a single atlas image (which translates to a single atlas within the TileSet). The unmerged atlases will be removed within the TileSet, but *the original tilesheet images will be kept on the filesystem*. If you don't want the unmerged atlases to be removed from the TileSet resource, choose **Merge (Keep Original Atlases)** instead.

Tip

TileSet features a system of *tile proxies*. Tile proxies are a mapping table that allows notifying the TileMap using a given TileSet that a given set of tile identifiers should be replaced by another one.

Tile proxies are automatically set up when merging different atlases, but they can also be set manually thanks to the **Manage Tile Proxies** dialog you can access using the "three vertical dots" menu mentioned above.

Manually creating tile proxies may be useful when you changed an atlas ID or want to replace all tiles from an atlas by the ones from another atlas. Note that when editing a TileMap, you can replace all cells by their corresponding mapped value.

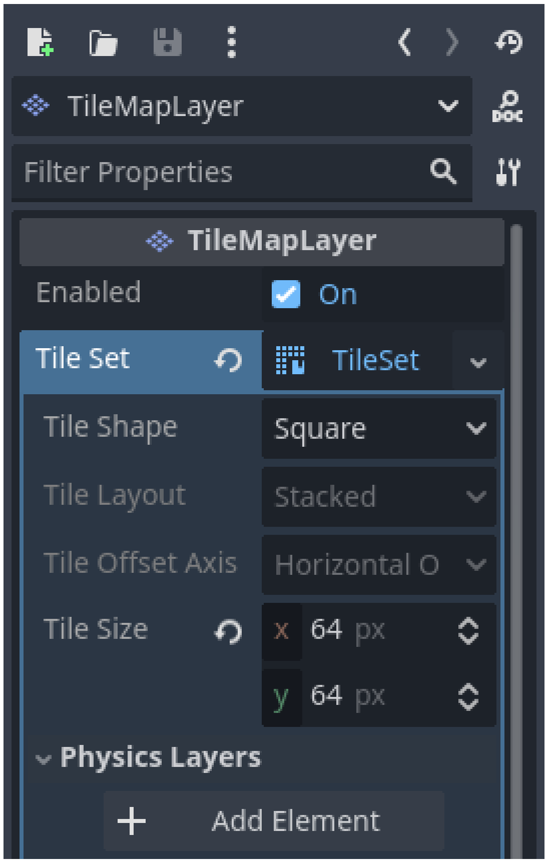
Adding collision, navigation and occlusion to the TileSet

We've now successfully created a basic TileSet. We could start using in the TileMapLayer node now, but it currently lacks any form of collision detection. This means the player and other objects could walk straight through the floor or walls.

If you use 2D navigation, you'll also need to define navigation polygons for tiles to generate a navigation mesh that agents can use for pathfinding.

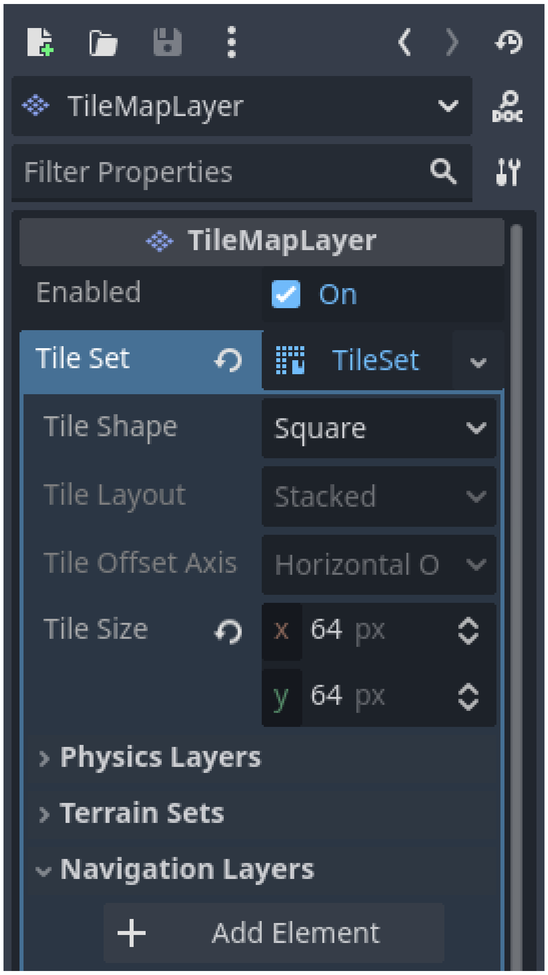
Lastly, if you use 2D lights and shadows or GPUParticles2D, you may also want your TileSet to be able to cast shadows and collide with particles. This requires defining occluder polygons for "solid" tiles on the TileSet.

To be able to define collision, navigation and occlusion shapes for each tile, you will need to create a physics, navigation or occlusion layer for the TileSet resource first. To do so, select the TileMapLayer node, click the TileSet property value in the inspector to edit it then unfold **Physics Layers** and choose **Add Element**:


TileMapLayer
Filter Properties
TileMapLayer
Enabled
On
Tile Set
TileSet
Tile Shape
Square
Tile Layout
Stacked
Tile Offset Axis
Horizontal O
Tile Size
x 64 px
y 64 px
Physics Layers
Add Element

Creating a physics layer in the TileSet resource inspector (within the TileMapLayer node)

If you also need navigation support, now is a good time to create a navigation layer:


TileMapLayer

Filter Properties

TileMapLayer

Enabled: On

Tile Set: TileSet

Tile Shape: Square

Tile Layout: Stacked

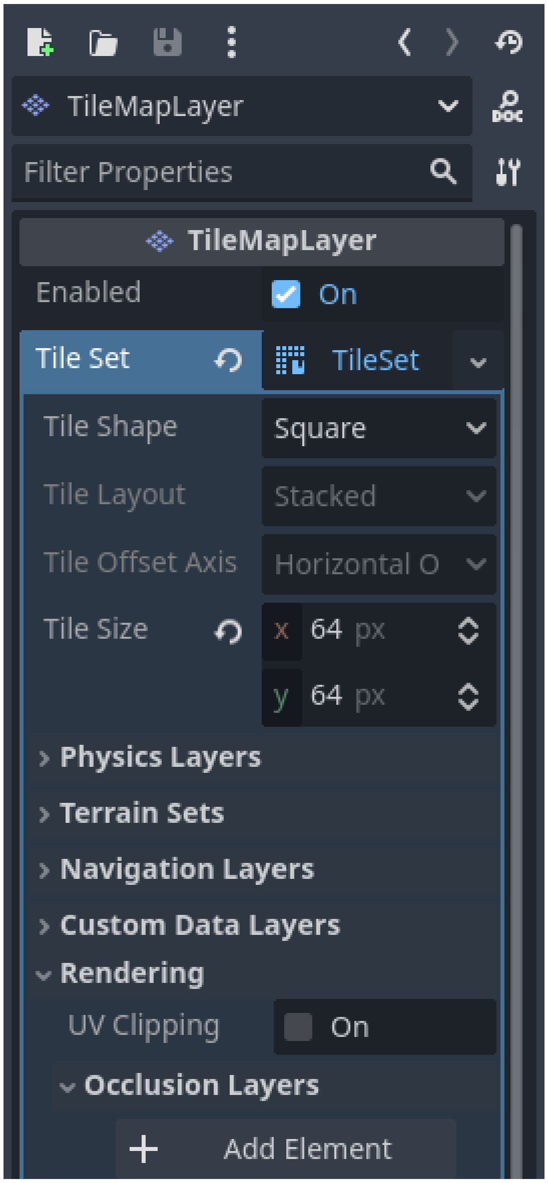
Tile Offset Axis: Horizontal O

Tile Size: x 64 px, y 64 px

Physics Layers
Terrain Sets
Navigation Layers
Add Element

Creating a navigation layer in the TileSet resource inspector (within the TileMapLayer node)

If you need support for light polygon occluders, now is a good time to create an occlusion layer:


TileMapLayer
Filter Properties
TileMapLayer
Enabled ☑ On
Tile Set TileSet
Tile Shape Square
Tile Layout Stacked
Tile Offset Axis Horizontal O
Tile Size x 64 px y 64 px
Physics Layers
Terrain Sets
Navigation Layers
Custom Data Layers
Rendering
UV Clipping ☐ On
Occlusion Layers
Add Element

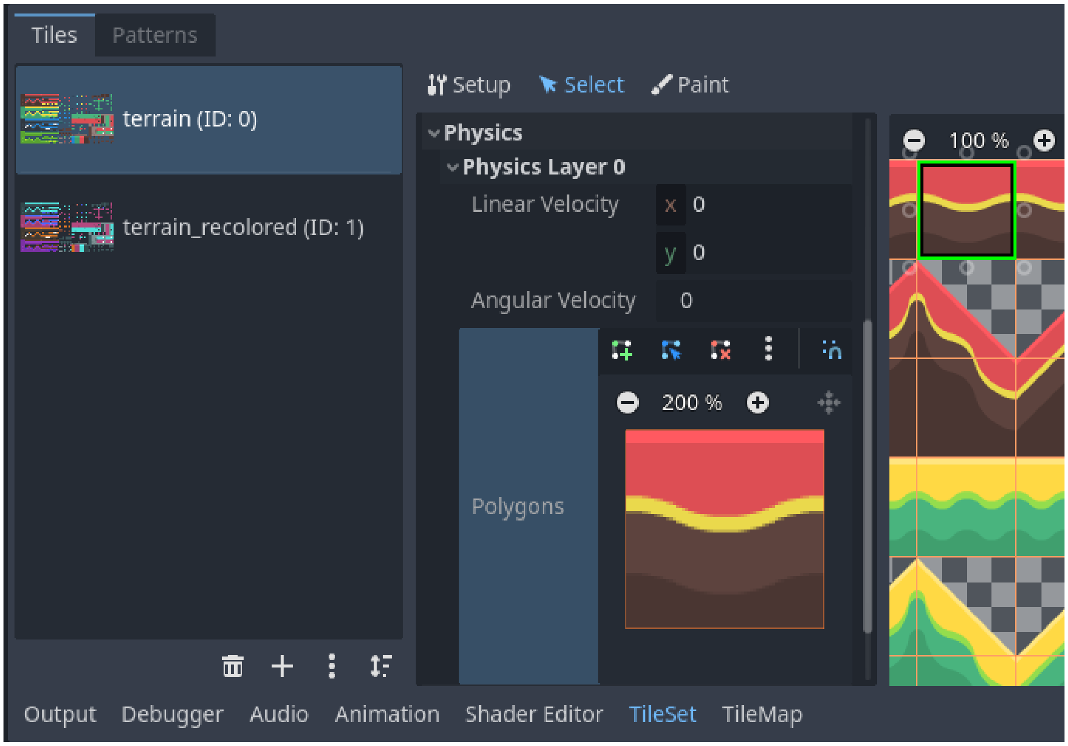
Creating an occlusion layer in the TileSet resource inspector (within the TileMapLayer node)

Note

Future steps in this tutorial are tailored to creating collision polygons, but the procedure for navigation and occlusion is very similar. Their respective polygon editors behave in the same way, so these steps are not repeated for brevity.

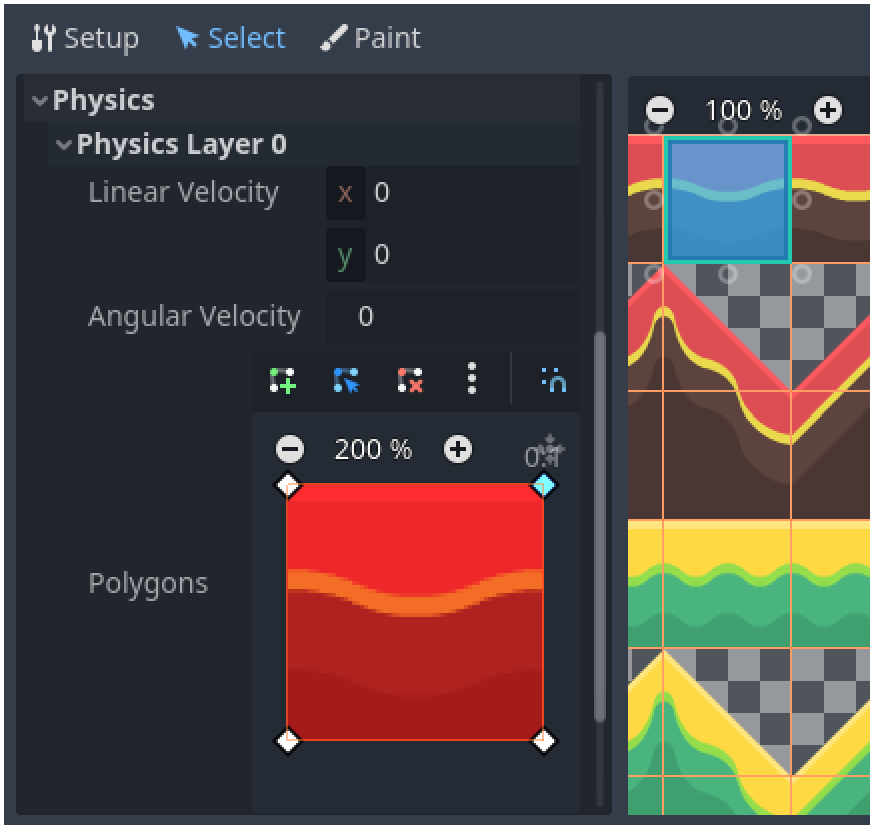
The only caveat is that the tile's occlusion polygon property is part of a **Rendering** subsection in the atlas inspector. Make sure to unfold this section so you can edit the polygon.

After creating a physics layer, you have access to the **Physics Layer** section in the TileSet atlas inspector:


Tiles
Patterns
terrain (ID: 0)
terrain_recolored (ID: 1)
Setup
Select
Paint
Physics
Physics Layer 0
Linear Velocity
x 0
y 0
Angular Velocity
0
Polygons
200 %
100 %
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Opening the collision editor while in Select mode

You can quickly create a rectangle collision shape by pressing `F` while the TileSet editor is focused. If the keyboard shortcut doesn't work, try clicking in the empty area around the polygon editor to focus it:

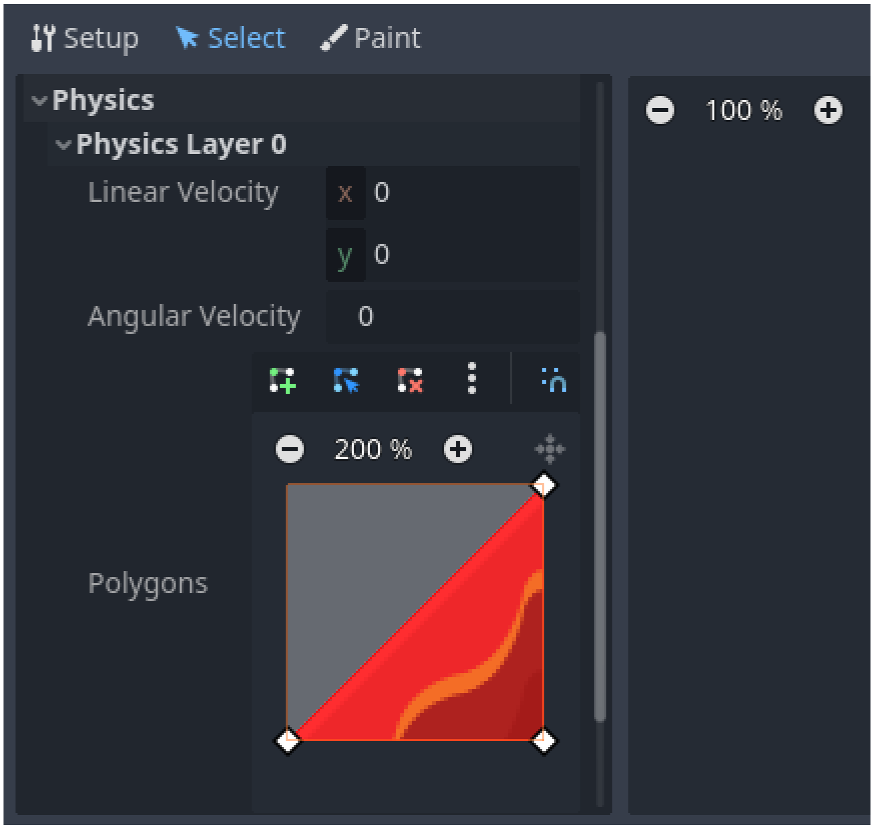

Setup
Select
Paint
Physics
Physics Layer 0
Linear Velocity
x 0
y 0
Angular Velocity
0
200 %
Polygons
100 %

Using default rectangle collision shape by pressing `F`

In this tile collision editor, you have access to all the 2D polygon editing tools:

- Use the toolbar above the polygon to toggle between creating a new polygon, editing an existing polygon and removing points on the polygon. The "three vertical dots" menu button offers additional options, such as rotating and flipping the polygon.
- Create new points by clicking and dragging a line between two points.
- Remove a point by right-clicking it (or using the Remove tool described above and left-clicking).
- Pan in the editor by middle-clicking or right-clicking. (Right-click panning can only be used in areas where there is no point nearby.)

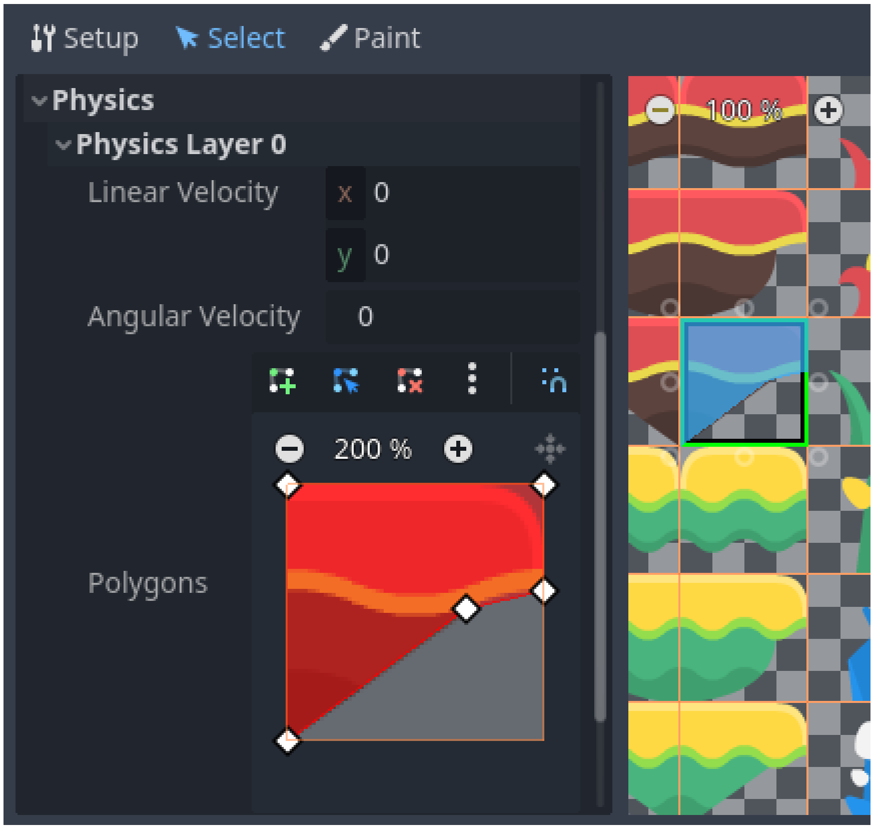
You can use the default rectangle shape to quickly create a triangle-shaped collision shape by removing one of the points:


Setup
Select
Paint
Physics
Physics Layer 0
100 %
200 %
Polygons

Linear Velocity x 0
y 0
Angular Velocity 0

Creating a triangle collision shape by right-clicking one of the corners to remove it

You can also use the rectangle as a base for more complex shapes by adding more points:



Physics

Physics Layer 0

Linear Velocity x 0 y 0

Angular Velocity 0

Polygons

Drawing a custom collision for a complex tile shape

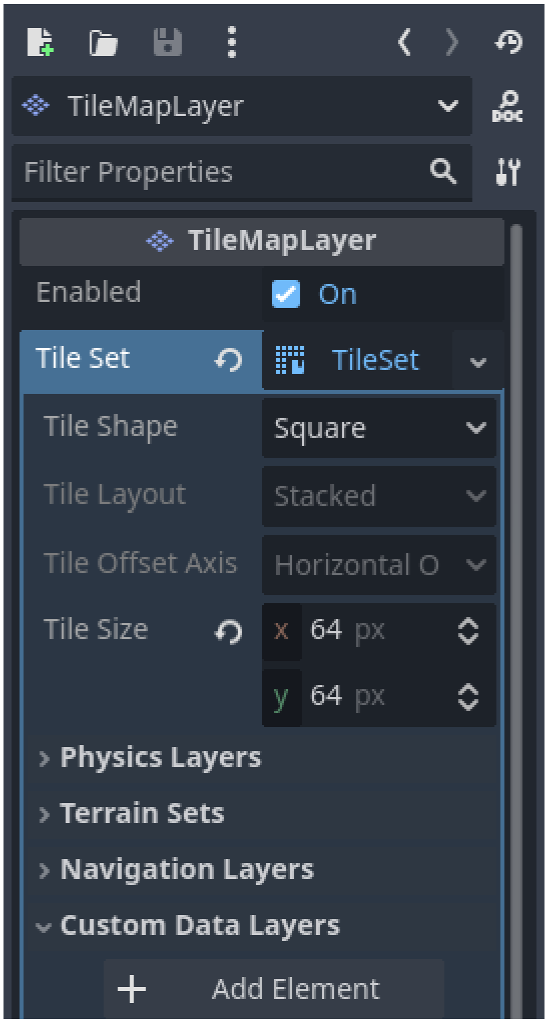
Tip

If you have a large tileset, specifying the collision for each tile individually could take a lot of time. This is especially true as TileMaps tend to have many tiles with common collision patterns (such as solid blocks or 45-degree slopes). To apply a similar collision shape to several tiles quickly, use functionality to [assign properties to multiple tiles at once]().

Assigning custom metadata to the TileSet's tiles

You can assign custom data on a per-tile basis using *custom data layers*. This can be useful to store information specific to your game, such as the damage that a tile should deal when the player touches it, or whether a tile can be destroyed using a weapon.

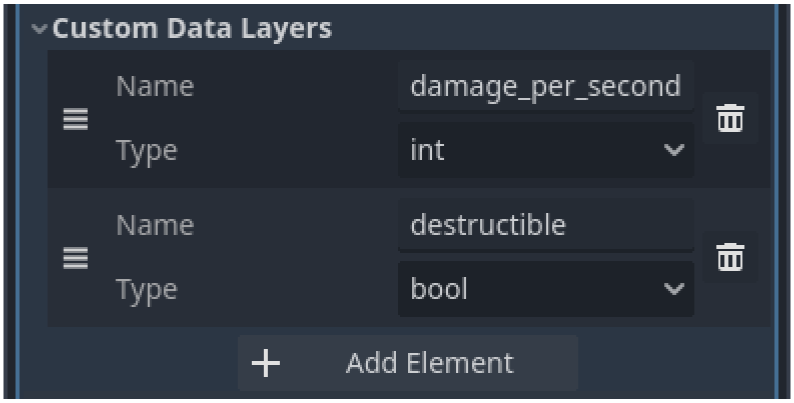
The data is associated with the tile in the TileSet: all instances of the placed tile will use the same custom data. If you need to create a variant of a tile that has different custom data, this can be done by [creating an alternative tile]() and changing the custom data for the alternative tile only.



Enabled: On
Tile Set: TileSet
Tile Shape: Square
Tile Layout: Stacked
Tile Offset Axis: Horizontal O
Tile Size: x 64 px, y 64 px

TileMapLayer
Filter Properties
TileMapLayer
Physics Layers
Terrain Sets
Navigation Layers
Custom Data Layers
Add Element

Creating a custom data layer in the TileSet resource inspector (within the TileMapLayer node)



Custom Data Layers

Name: damage_per_second
Type: int

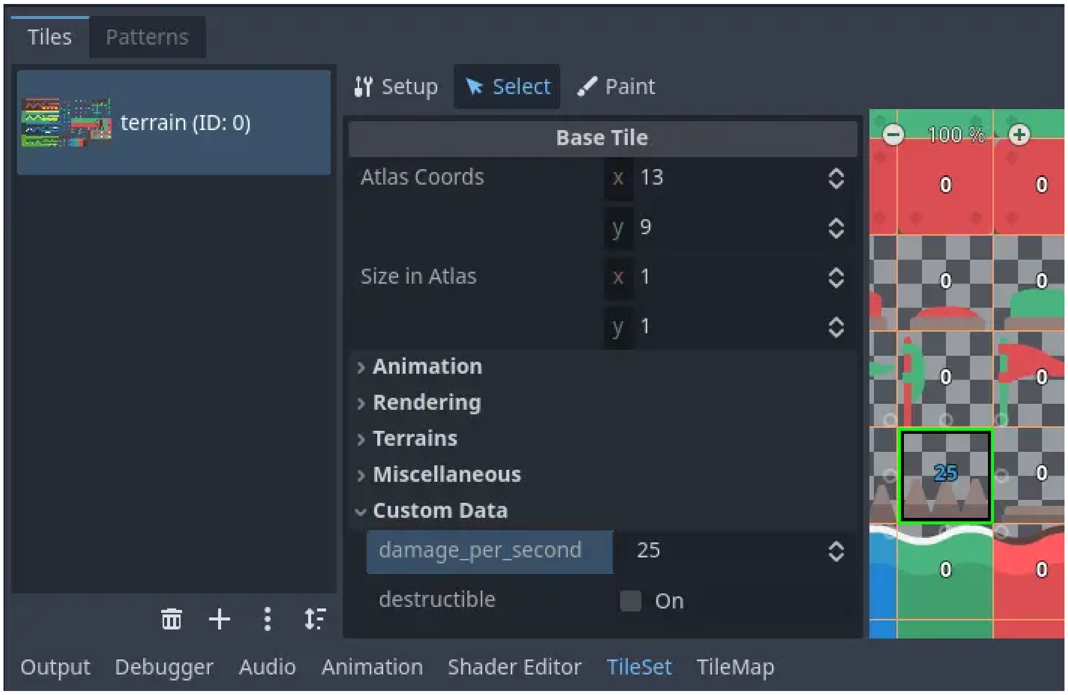
Name: destructible
Type: bool

Add Element

Example of configured custom data layers with game-specific properties

You can reorder custom data without breaking existing metadata: the TileSet editor will update automatically after reordering custom data properties.

With the custom data layers example shown above, we're assigning a tile to have the `damage_per_second` metadata set to `25` and the `destructible` metadata to `false`:



Base Tile

Atlas Coords: x 13, y 9
Size in Atlas: x 1, y 1

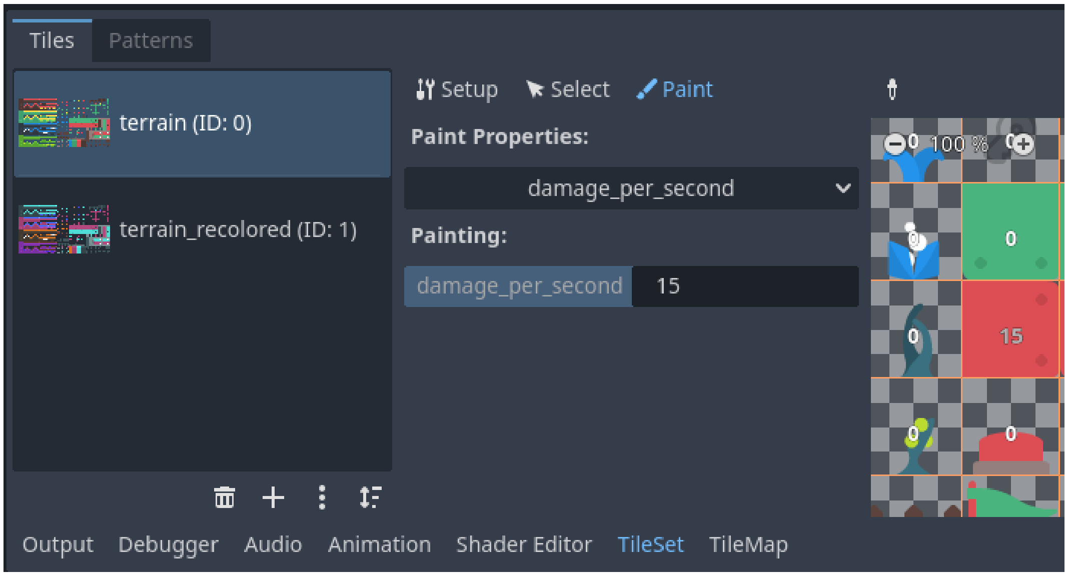
Custom Data

damage_per_second: 25
destructible: ☐ On

Tiles
Patterns
terrain (ID: 0)
Setup
Select
Paint
Animation
Rendering
Terrains
Miscellaneous
100 %
0
25
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Editing custom data in the TileSet editor while in Select mode

Tile property painting can also be used for custom data:


Tiles
Patterns
terrain (ID: 0)
terrain_recolored (ID: 1)
Setup
Select
Paint
100 %
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap
0
0
15
0
0
0
0

Paint Properties: damage_per_second

Painting: damage_per_second 15

Assigning custom data in the TileSet editor using tile property painting

Creating terrain sets (autotiling)

Note

This functionality was implemented in a different form as *autotiling* in Godot 3.x. Terrains are essentially a more powerful replacement of autotiles. Unlike autotiles, terrains can support transitions from one terrain to another, as a tile may define several terrains at once.

Unlike before, where autotiles were a specific kind of tiles, terrains are only a set of properties assigned to atlas tiles. These properties are then used by a dedicated TileMap painting mode that selects tiles featuring terrain data in a smart way. This means any terrain tile can be either painted as terrain or as a single tile, like any other.

A "polished" tileset generally features variations that you should use on corners or edges of platforms, floors, etc. While these can be placed manually, this quickly becomes tedious. Handling this situation with procedurally generated levels can also be difficult and require a lot of code.

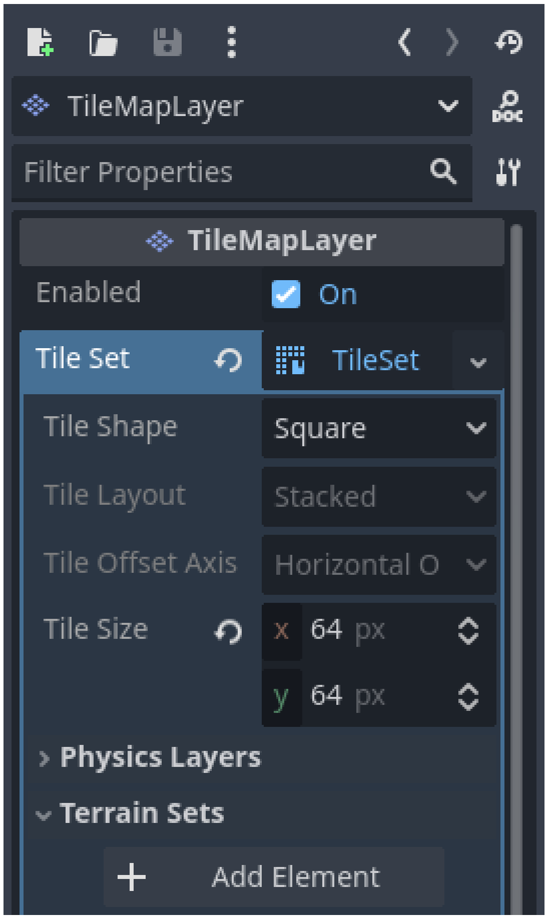
Godot offers *terrains* to perform this kind of tile connections automatically. This allows you to have the "correct" tile variants automatically used.

Terrains are grouped into terrain sets. Each terrain set is assigned a mode from **Match Corners and Sides**, **Match Corners** and **Match sides**. They define how terrains are matched to each other in a terrain set.

Note

The above modes correspond to the previous bitmask modes autotiles used in Godot 3.x: 2×2, 3×3 or 3×3 minimal. This is also similar to what the Tiled [https://www.mapeditor.org/] editor features.

Select the TileMapLayer node, go to the inspector and create a new terrain set within the TileSet *resource*:


TileMapLayer
Filter Properties
TileMapLayer
Physics Layers
Terrain Sets
Add Element

Enabled: On

Tile Set: TileSet

Tile Shape: Square

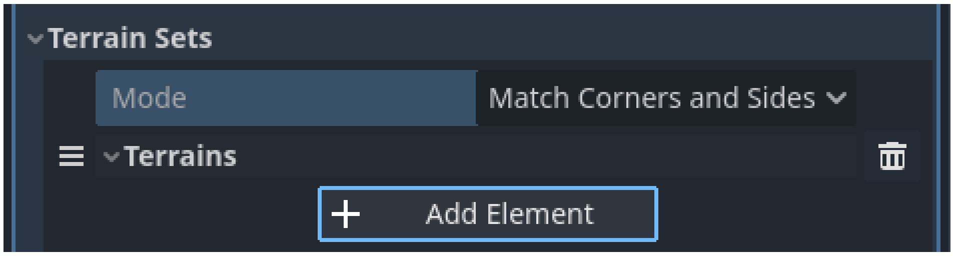
Tile Layout: Stacked

Tile Offset Axis: Horizontal O

Tile Size: x 64 px, y 64 px

Creating a terrain set in the TileSet resource inspector (within the TileMapLayer node)

After creating a terrain set, you **must** create one or more terrains *within* the terrain set:


Terrain Sets
Terrains
Add Element

Mode: Match Corners and Sides

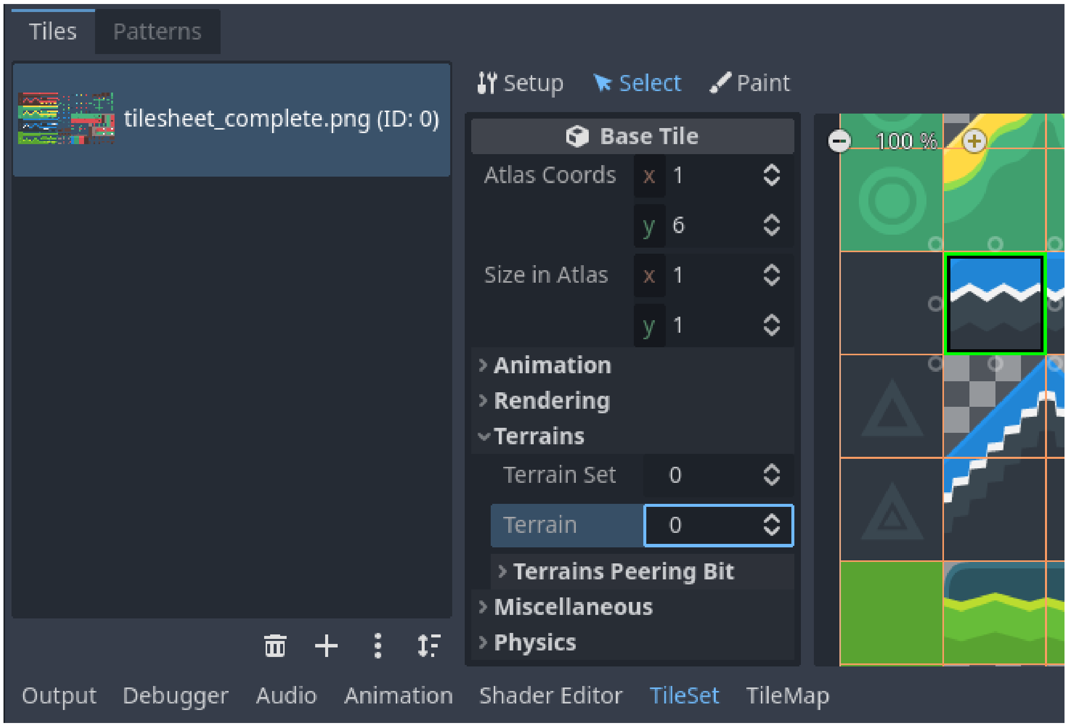
Creating a terrain within the terrain set

In the TileSet editor, switch to Select mode and click a tile. In the middle column, unfold the **Terrains** section then assign a terrain set

ID and a terrain ID for the tile. `-1` means "no terrain set" or "no terrain", which means you must set **Terrain Set** to `0` or greater before you can set **Terrain** to `0` or greater.

Note

Terrain set IDs and terrain IDs are independent from each other. They also start from `0`, not `1`.


Tiles
Patterns
tilesheet_complete.png (ID: 0)
Setup
Select
Paint
Base Tile
100 %
Animation
Rendering
Terrains
Terrains Peering Bit
Miscellaneous
Physics
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Atlas Coords x 1 y 6

Size in Atlas x 1 y 1

Terrain Set 0

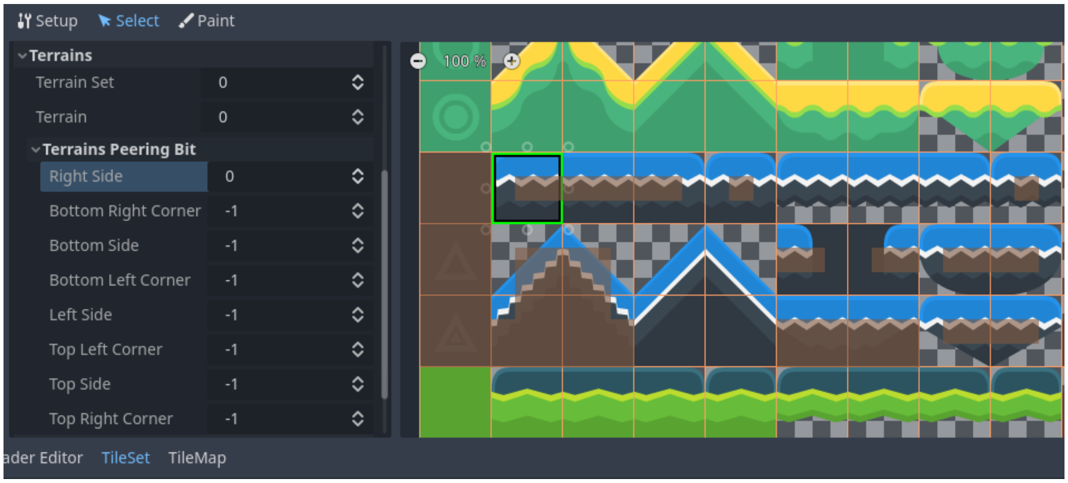
Terrain 0

Configuring terrain on a single tile in the TileSet editor's Select mode

After doing so, you can now configure the **Terrain Peering Bits** section which becomes visible in the middle column. The peering bits determine which tile will be placed depending on neighboring tiles. `-1` is a special value which refers to empty space.

For example, if a tile has all its bits set to `0` or greater, it will only appear if *all* 8 neighboring tiles are using a tile with the same terrain

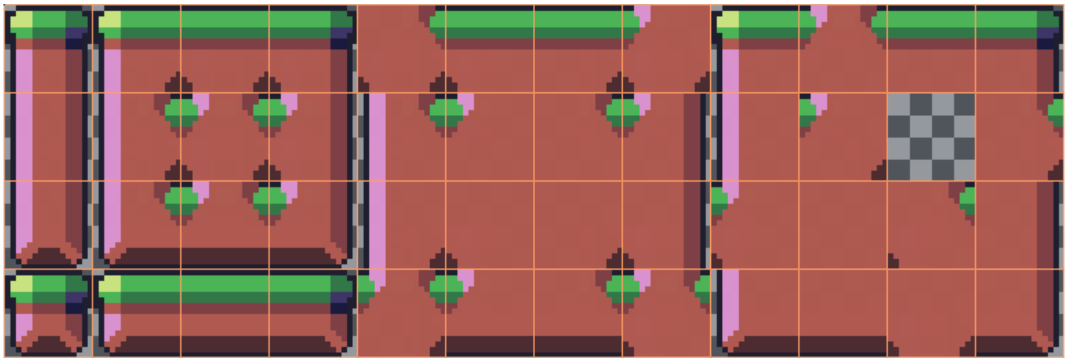
ID. If a tile has its bits set to `0` or greater, but the top-left, top and top-right bits are set to `-1`, it will only appear if there is empty space on top of it (including diagonally).



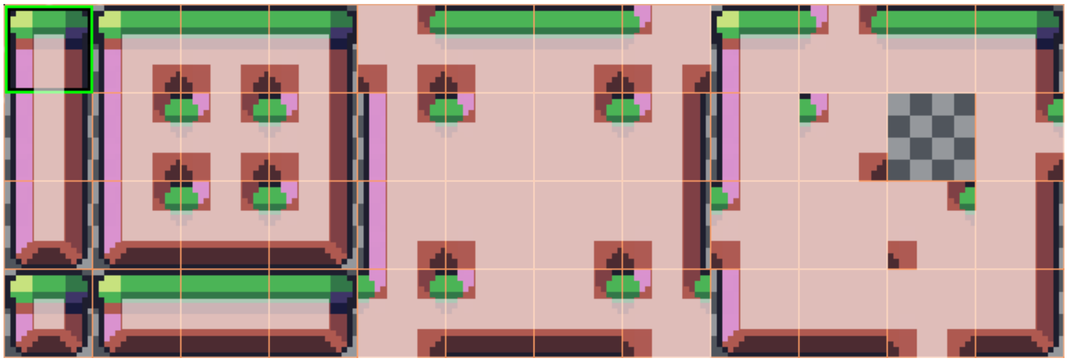
Terrains
Terrain Set 0
Terrain 0
Terrains Peering Bit
Right Side 0
Bottom Right Corner -1
Bottom Side -1
Bottom Left Corner -1
Left Side -1
Top Left Corner -1
Top Side -1
Top Right Corner -1

Configuring terrain peering bits on a single tile in the TileSet editor's Select mode

An example configuration for a full tilesheet may look as follows:



Example full tilesheet for a sidescrolling game



Example full tilesheet for a sidescrolling game with terrain peering bits visible

Assigning properties to multiple tiles at once

There are two ways to assign properties to multiple tiles at once. Depending on your use cases, one method may be faster than the other:

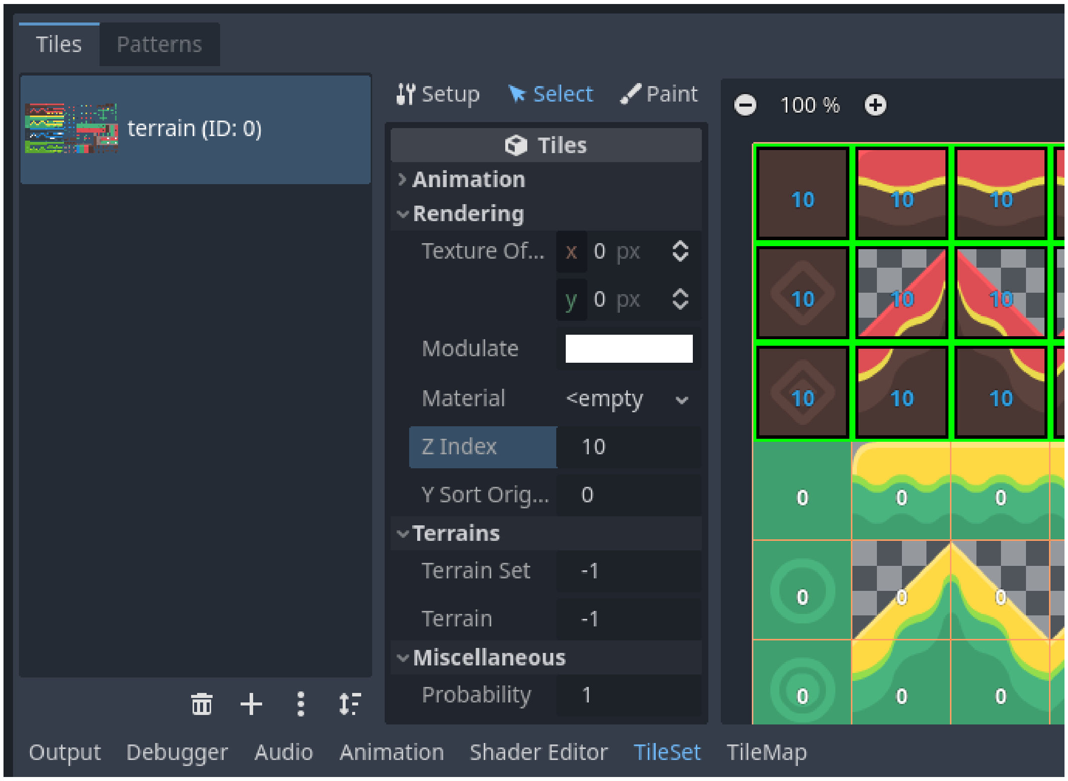
Using multiple tile selection

If you wish to configure various properties on several tiles at once, choose the **Select** mode at the top of the TileSet editor:

After doing this, you can select multiple tiles on the right column by holding `Shift` then clicking on tiles. You can also perform rectangle selection by holding down the left mouse button then dragging the mouse. Lastly, you can deselect tiles that were already selected (without affecting the rest of the selection) by holding `Shift` then clicking on a selected tile.

You can then assign properties using the inspector in the middle column of the TileSet editor. Only properties that you change here will be applied to all selected tiles. Like in the editor's inspector, properties that differ on selected tiles will remain different until you edit them.

With numerical and color properties, you will also see a preview of the property's value on all tiles in the atlas after editing a property:


Tiles
Patterns
terrain (ID: 0)
Setup
Select
Paint
100 %
Tiles
Animation
Rendering
Terrains
Miscellaneous
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap
10
0

Texture Of... x 0 px y 0 px

Modulate

Material <empty

Z Index 10

Y Sort Orig... 0

Terrain Set -1

Terrain -1

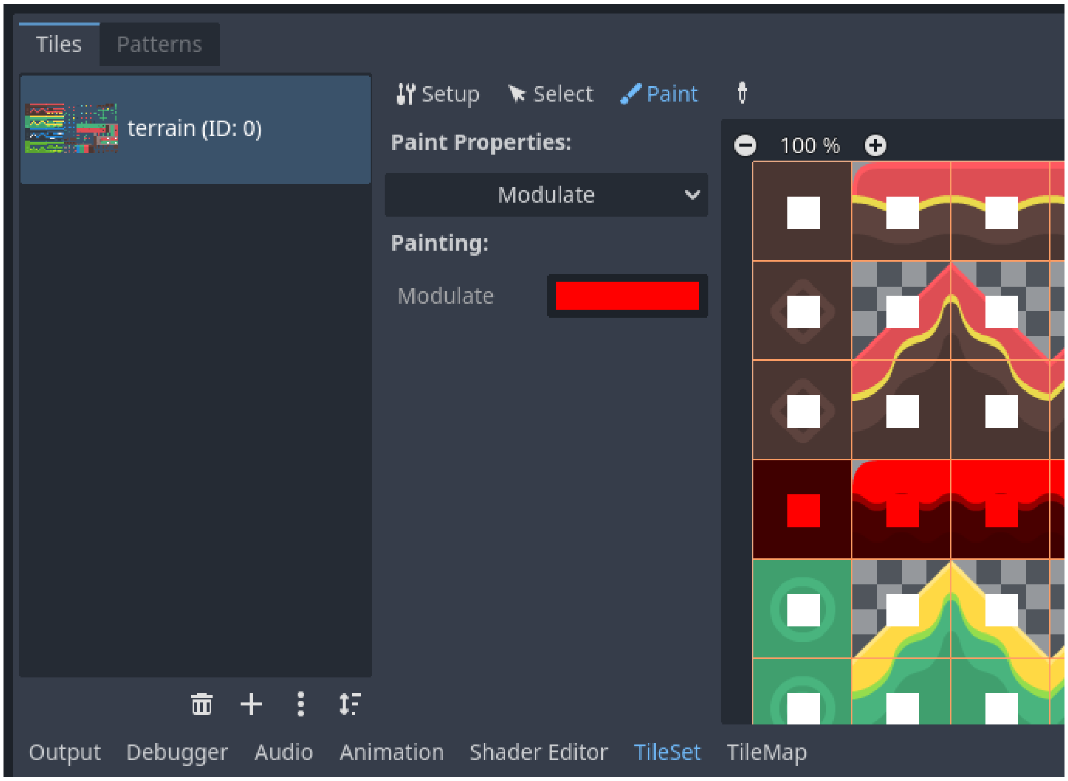
Probability 1

Selecting multiple tiles using the Select mode, then applying properties

Using tile property painting

If you wish to apply a single property to several tiles at once, you can use the *property painting* mode for this purpose.

Configure a property to be painted in the middle column, then click on tiles (or hold down the left mouse button) in the right column to "paint" properties onto tiles.


Tiles
Patterns
terrain (ID: 0)
Setup
Select
Paint
100 %
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Paint Properties:

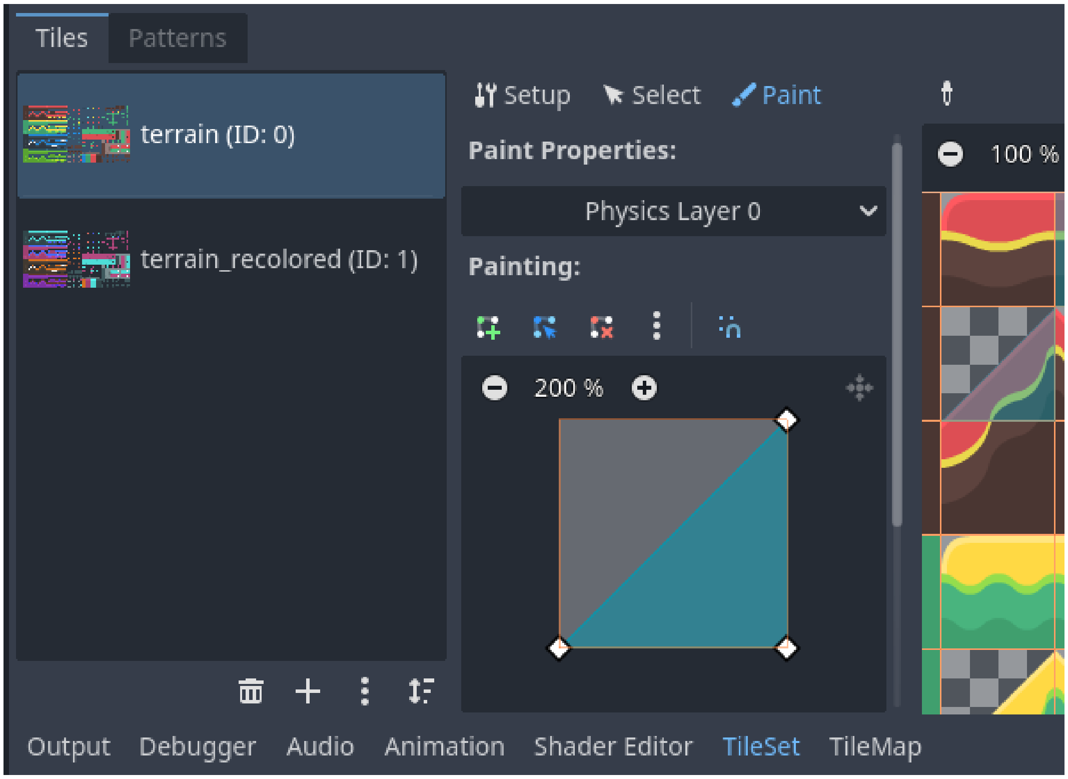
Modulate

Painting:

Modulate

Painting tile properties using the TileSet editor

Tile property painting is especially useful with properties that are time-consuming to set manually, such as collision shapes:


Tiles
Patterns
terrain (ID: 0)
terrain_recolored (ID: 1)
Setup
Select
Paint
Paint Properties:
Physics Layer 0
Painting:
200 %
100 %
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Painting a collision polygon, then left-clicking tiles to apply it

Creating alternative tiles

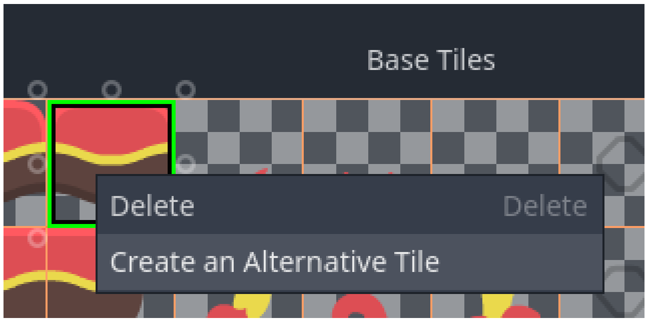
Sometimes, you want to use a single tile image (found only once within the atlas), but configured in different ways. For example, you may want to use the same tile image, but rotated, flipped, or modulated with a different color. This can be done using *alternative tiles*.

Tip

Since Godot 4.2, you don't have to create alternative tiles to rotate or flip tiles anymore. You can rotate any tile while placing it in the

TileMap editor by using the rotation/flip buttons in the TileMap editor toolbar.

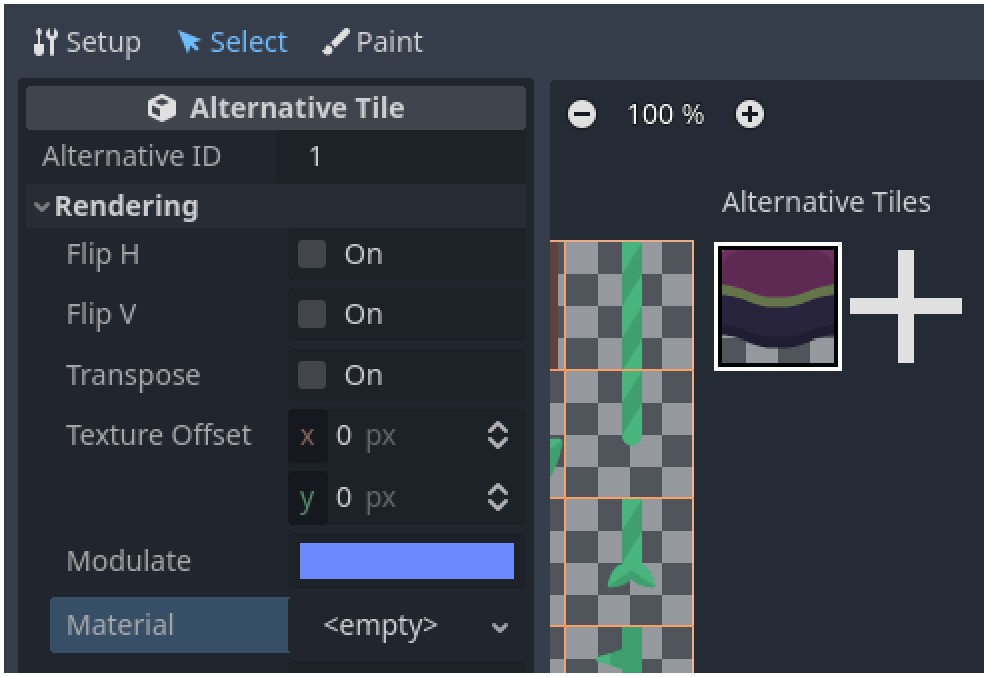
To create an alternative tile, right-click a base tile in the atlas displayed by the TileSet editor, then choose **Create an Alternative Tile**:



Creating an alternative tile by right-clicking a base tile in the TileSet editor

If currently in Select mode, the alternative tile will already be selected for editing. If not currently in Select mode, you can still create alternative tiles, but you will need to switch to Select mode and select the alternative tile to edit it.

If you don't see the alternative tile, pan over to the right of the atlas image, as alternative tiles always appear on the right of base tiles of a given atlas in the TileSet editor:



Alternative ID 1

Rendering

Flip H ☐ On

Flip V ☐ On

Transpose ☐ On

Texture Offset x 0 px y 0 px

Modulate

Material <empty>

Configuring an alternative tile after clicking it in the TileSet editor

After selecting an alternative tile, you can change any properties using the middle column like you would on a base tile. However, the list of exposed properties is different compared to base tiles:

- **Alternative ID:** The unique numerical identifier for this alternative tile. Changing it will break existing TileMaps, so be careful! This ID also controls the sorting in the list of alternative tiles displayed in the editor.
- **Rendering > Flip H:** If `true`, the tile is horizontally flipped.
- **Rendering > Flip V:** If `true`, the tile is vertically flipped.
- **Rendering > Transpose:** If `true`, the tile is rotated 90 degrees *counter-clockwise* and then flipped vertically. In practice, this means that to rotate a tile by 90 degrees clockwise without flipping it, you should enable **Flip H** and **Transpose**. To rotate a tile by 180 degrees clockwise, enable **Flip H** and **Flip V**. To rotate a tile by 270 degrees clockwise, enable **Flip V** and **Transpose**.

- **Rendering > Texture Origin:** The origin to use for drawing the tile. This can be used to visually offset the tile compared to the base tile.
- **Rendering > Modulate:** The color multiplier to use when rendering the tile.
- **Rendering > Material:** The material to use for this tile. This can be used to apply a different blend mode or custom shaders to a single tile.
- **Z Index:** The sorting order for this tile. Higher values will make the tile render in front of others on the same layer.
- **Y Sort Origin:** The vertical offset to use for tile sorting based on its Y coordinate (in pixels). This allows using layers as if they were on different height for top-down games. Adjusting this can help alleviate issues with sorting certain tiles. Only effective if **Y Sort Enabled** is `true` on the TileMapLayer node under **CanvasItem > Ordering**

You can create an additional alternative tile variant by clicking the large "+" icon next to the alternative tile. This is equivalent to selecting the base tile and right-clicking it to choose **Create an Alternative Tile** again.

Note

When creating an alternative tile, none of the properties from the base tile are inherited. You must set properties again on the alternative tile if you wish those to be identical on the base tile and the alternative tile.

Using TileMaps

See also

This page assumes you have created or downloaded a TileSet already. If not, please read Using TileSets first as you will need a TileSet to create a TileMap.

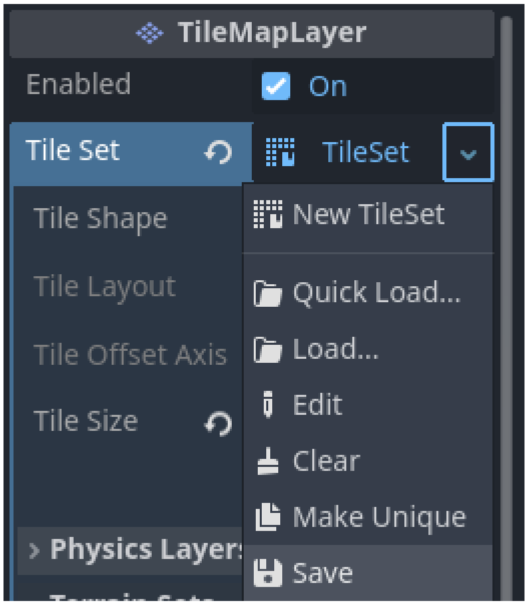
Introduction

A tilemap is a grid of tiles used to create a game's layout. There are several benefits to using TileMapLayer nodes to design your levels. First, they make it possible to draw the layout by "painting" the tiles onto a grid, which is much faster than placing individual Sprite2D nodes one by one. Second, they allow for much larger levels because they are optimized for drawing large numbers of tiles. Finally, you can add collision, occlusion, and navigation shapes to tiles, adding greater functionality to the TileMap.

Specifying the TileSet in the TileMapLayer

If you've followed the previous page on Using TileSets, you should have a TileSet resource that is built-in to the TileMapLayer node. This is good for prototyping, but in a real world project, you will generally have multiple levels reusing the same tileset.

The recommended way to reuse the same TileSet in several TileMapLayer nodes is to save the TileSet to an external resource. To do so, click the dropdown next to the TileSet resource and choose **Save**:


TileMapLayer
Enabled
On
Tile Set
TileSet
Tile Shape
Tile Layout
Tile Offset Axis
Tile Size
Physics Layers
New TileSet
Quick Load...
Load...
Edit
Clear
Make Unique
Save

Saving the built-in TileSet resource to an external resource file

Multiple TileMapLayers and settings

When working with tilemaps it's generally advised that you use multiple TileMapLayer nodes when appropriate. Using multiple layers can be advantageous, for example, this allows you to distinguish foreground tiles from background tiles for better organization. You can place one tile per layer at a given location, which allows you to overlap several tiles together if you have more than one layer.

Each TileMapLayer node has several properties you can adjust:

- **Enabled:** If `true`, the layer is visible in the editor and when running the project.
- **TileSet** The tileset used by the TileMapLayer node.

Rendering

- **Y Sort Origin:** The vertical offset to use for Y-sorting on each tile (in pixels). Only effective if **Y Sort Enabled** under CanvasItem settings is `true`.
- **X Draw Order Reversed** Reverses the order tiles are drawn on the X axis. Requires that **Y Sort Enabled** under CanvasItem settings is `true`.
- **Rendering Quadrant Size** A quadrant is a group of tiles drawn together on a single CanvasItem for optimization purposes. This setting defines the length of a square's side in the map's coordinate system. The quadrant size does not apply to a Y sorted TileMapLayer since tiles are grouped by Y position in that case.

Physics

- **Collision Enabled** Enables or disables collision.
- **Use Kinematic Bodies** When true TileMapLayer collision shapes will be instantiated as kinematic bodies.
- **Collision Visibility Mode** Whether or not the TileMapLayer's collision shapes are visible. If set to default, then it depends on the show collision debug settings.

Navigation

- **Navigation Enabled** Whether or not navigation regions are enabled.
- **Navigation Visible** Whether or not the TileMapLayer's navigation meshes are visible. If set to default then it depends on the show navigation debug settings.

Tip

TileMap built-in navigation has many practical limitations that result in inferior pathfinding performance and pathfollowing quality.

After designing the TileMap consider baking it to a more optimized navigation mesh (and disabling the TileMap NavigationLayer) using a NavigationRegion2D or the NavigationServer2D. See Using navigation meshes for additional information.

Warning

2D navigation meshes can not be "layered" or stacked on top of each other like visuals or physic shapes. Attempting to stack navigation meshes on the same navigation map will result in merge and logical errors that break the pathfinding.

Reordering layers

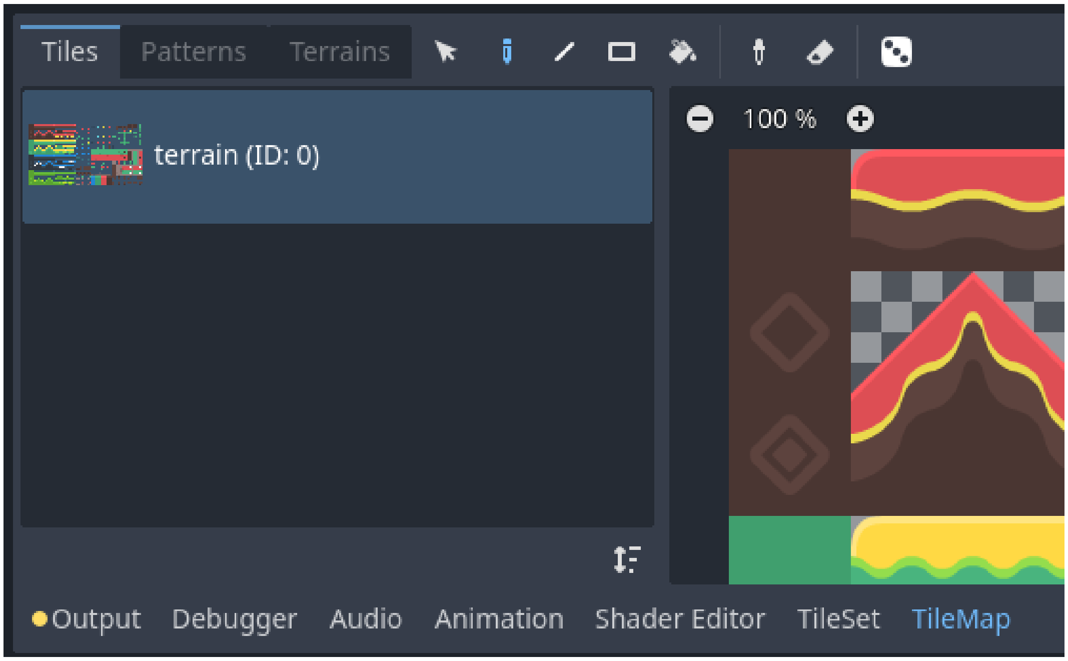
You can reorder layers by drag-and-dropping their node in the Scene tab. You can also switch between which TileMapLayer node you're working on by using the buttons in the top right corner of the TileMap editor.

Note

You can create, rename or reorder layers in the future without affecting existing tiles. Be careful though, as *removing* a layer will also remove all tiles that were placed on the layer.

Opening the TileMap editor

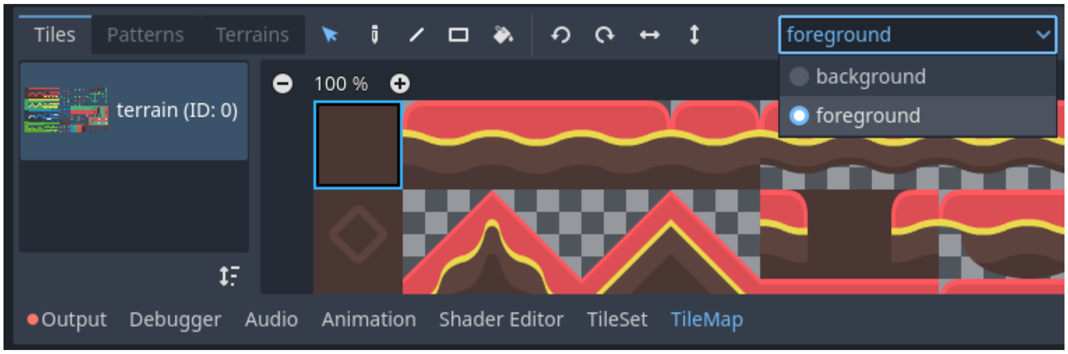
Select the TileMapLayer node, then open the TileMap panel at the bottom of the editor:


Tiles
Patterns
Terrains
terrain (ID: 0)
100 %
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Opening the TileMap panel at the bottom of the editor. The TileMapLayer node must be selected first.

Selecting tiles to use for painting

First, if you've created additional layers above, make sure you've selected the layer you wish to paint on:


Tiles
Patterns
Terrains
foreground
background
foreground
terrain (ID: 0)
100 %
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

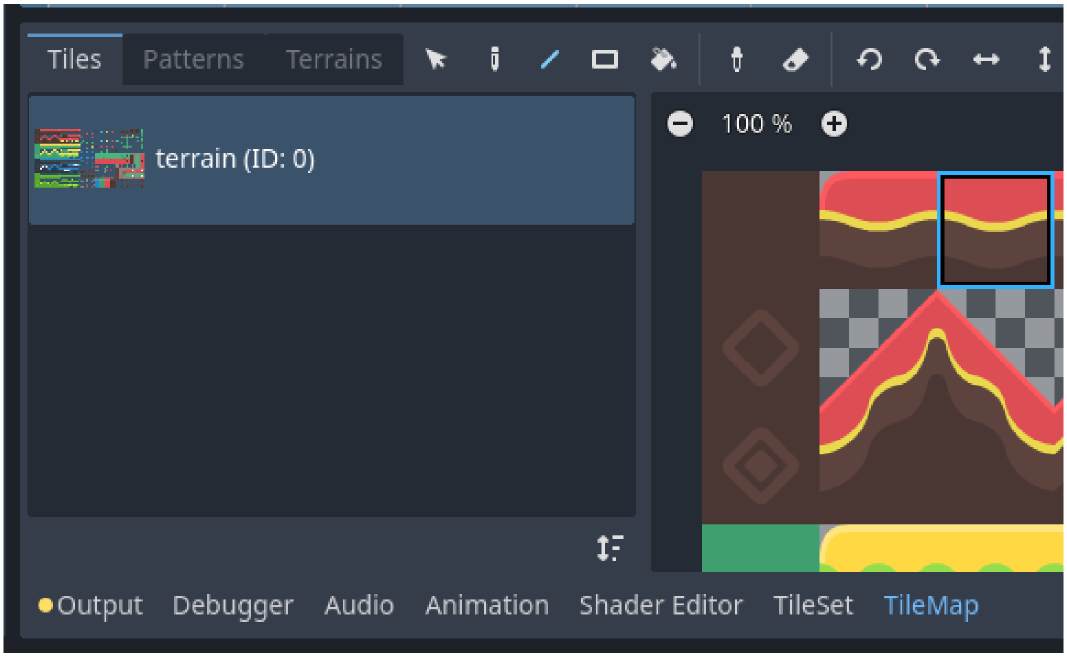
Selecting a layer to paint on in the TileMap editor

Tip

In the 2D editor, the layers you aren't currently editing from the same TileMapLayer node will appear grayed out while in the TileMap editor. You can disable this behavior by clicking the icon next to the layer selection menu (**Highlight Selected TileMap Layer** tooltip).

You can skip the above step if you haven't created additional layers, as the first layer is automatically selected when entering the TileMap editor.

Before you can place tiles in the 2D editor, you must select one or more tiles in the TileMap panel located at the bottom of the editor. To do so, click a tile in the TileMap panel, or hold down the mouse button to select multiple tiles:


Tiles
Patterns
Terrains
terrain (ID: 0)
100 %
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

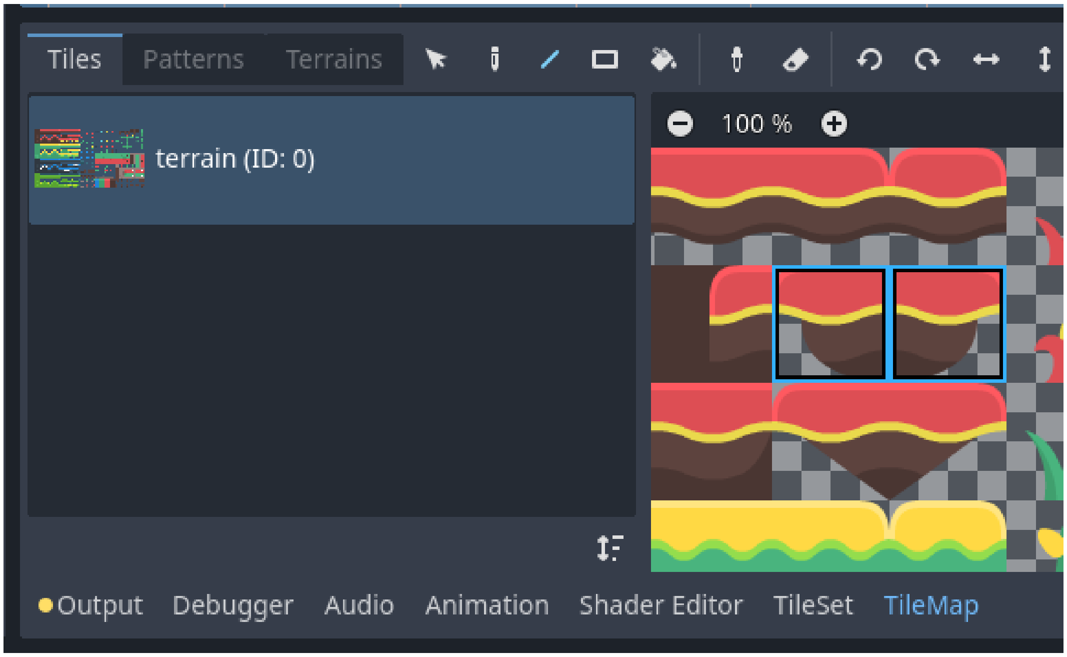
Selecting a tile in the TileMap editor by clicking it

Tip

Like in the 2D and TileSet editors, you can pan across the TileMap panel using the middle or right mouse buttons, and zoom using the mouse wheel or buttons in the top-left corner.

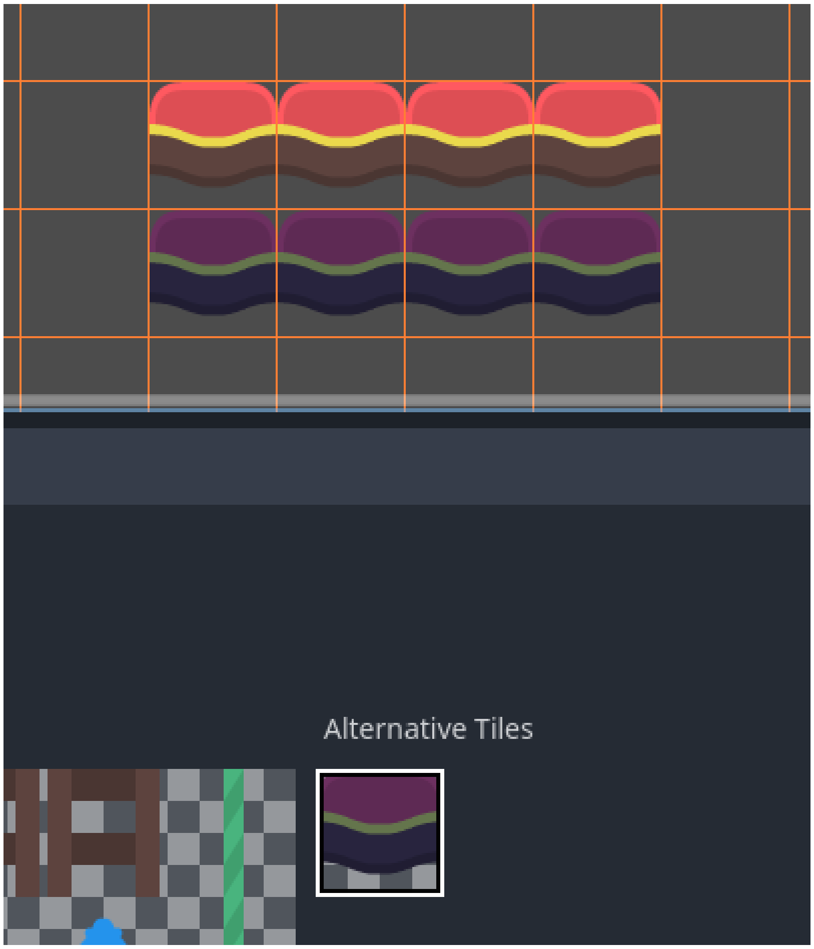
You can also hold down `Shift` to append to the current selection. When selecting more than one tile, multiple tiles will be placed every time you perform a painting operation. This can be used to paint structures composed of multiple tiles in a single click (such as large platforms or trees).

The final selection does not have to be contiguous: if there is empty space between selected tiles, it will be left empty in the pattern that will be painted in the 2D editor.


Tiles
Patterns
Terrains
terrain (ID: 0)
100 %
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

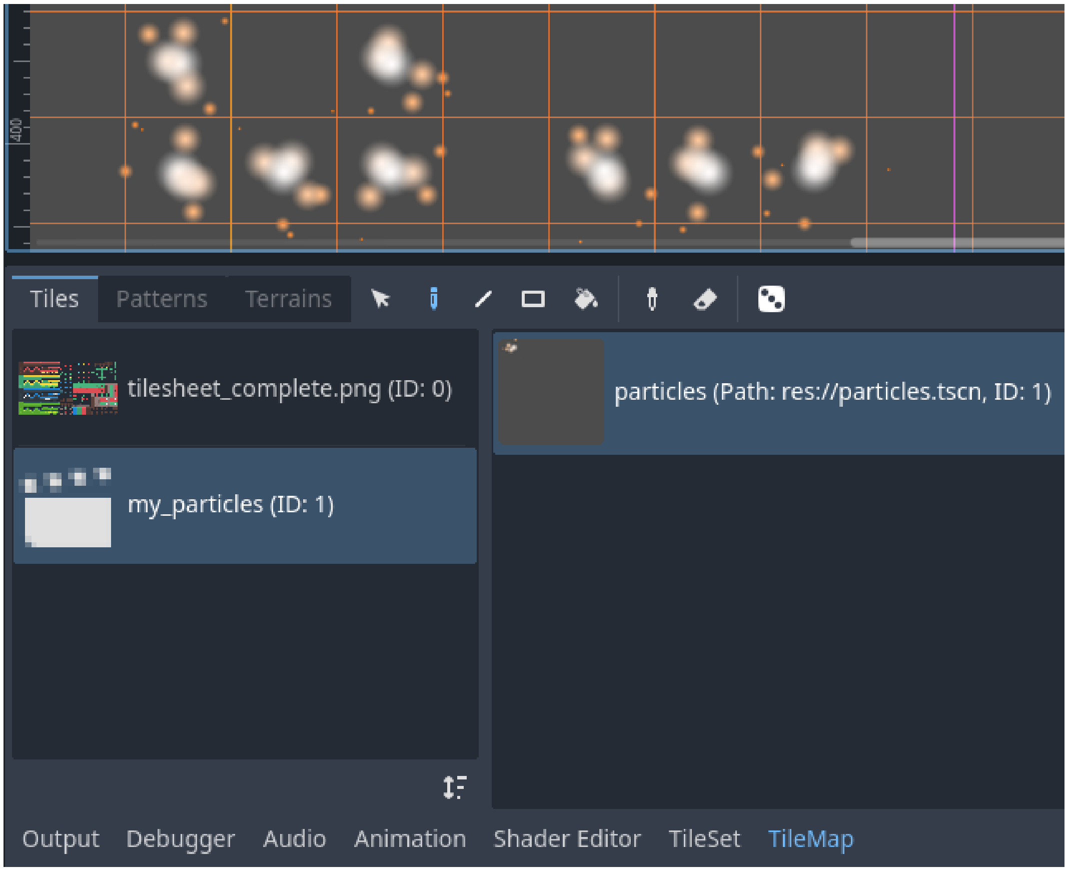
Selecting multiple tiles in the TileMap editor by holding down the left mouse button

If you've created alternative tiles in your TileSet, you can select them for painting on the right of the base tiles:


Alternative Tiles

Selecting an alternative tile in the TileMap editor

Lastly, if you've created a *scenes collection* in the TileSet, you can place scene tiles in the TileMap:


Tiles
Patterns
Terrains
tilesheet_complete.png (ID: 0)
my_particles (ID: 1)
particles (Path: res://particles.tscn, ID: 1)
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Placing a scene tile containing particles using the TileMap editor

Painting modes and tools

Using the toolbar at the top of the TileMap editor, you can choose between several painting modes and tools. These modes affect operation when clicking in the 2D editor, **not** the TileMap panel itself.

From left to right, the painting modes and tools you can choose are:

Selection

Select tiles by clicking a single tile, or by holding down the left mouse button to select multiple with a rectangle in the 2D editor. Note that empty space cannot be selected: if you create a rectangle selection, only non-empty tiles will be selected.

To append to the current selection, hold `Shift` then select a tile. To remove from the current selection, hold `Ctrl` then select a tile.

The selection can then be used in any other painting mode to quickly create copies of an already-placed pattern.

You can remove the selected tiles from the TileMap by pressing `Del`.

You can toggle this mode temporarily while in Paint mode by holding `Ctrl` then performing a selection.

Tip

You can copy and paste tiles that were already placed by performing a selection, pressing `Ctrl + C` then pressing `Ctrl + V`. The selection will be pasted after left-clicking. You can press `Ctrl + V` another time to perform more copies this way. Right-click or press `Escape` to cancel pasting.

Paint

The standard Paint mode allows you to place tiles by clicking or holding down the left mouse button.

If you right-click, the currently selected tile will be erased from the tilemap. In other words, it will be replaced by empty space.

If you have selected multiple tiles in the TileMap or using the Selection tool, they will be placed every time you click or drag the mouse while holding down the left mouse button.

Tip

While in Paint mode, you can draw a line by holding `Shift` *before* holding down the left mouse button, then dragging the mouse to the line's end point. This is identical to using the Line tool described below.

You can also draw a rectangle by holding `Ctrl` and `Shift` *before* holding down the left mouse button, then dragging the mouse to the rectangle's end point. This is identical to using the Rectangle tool described below.

Lastly, you can pick existing tiles in the 2D editor by holding `Ctrl` then clicking on a tile (or holding and dragging the mouse). This will switch the currently painted tile(s) to the tile(s) you've just clicked. This is identical to using the Picker tool described below.

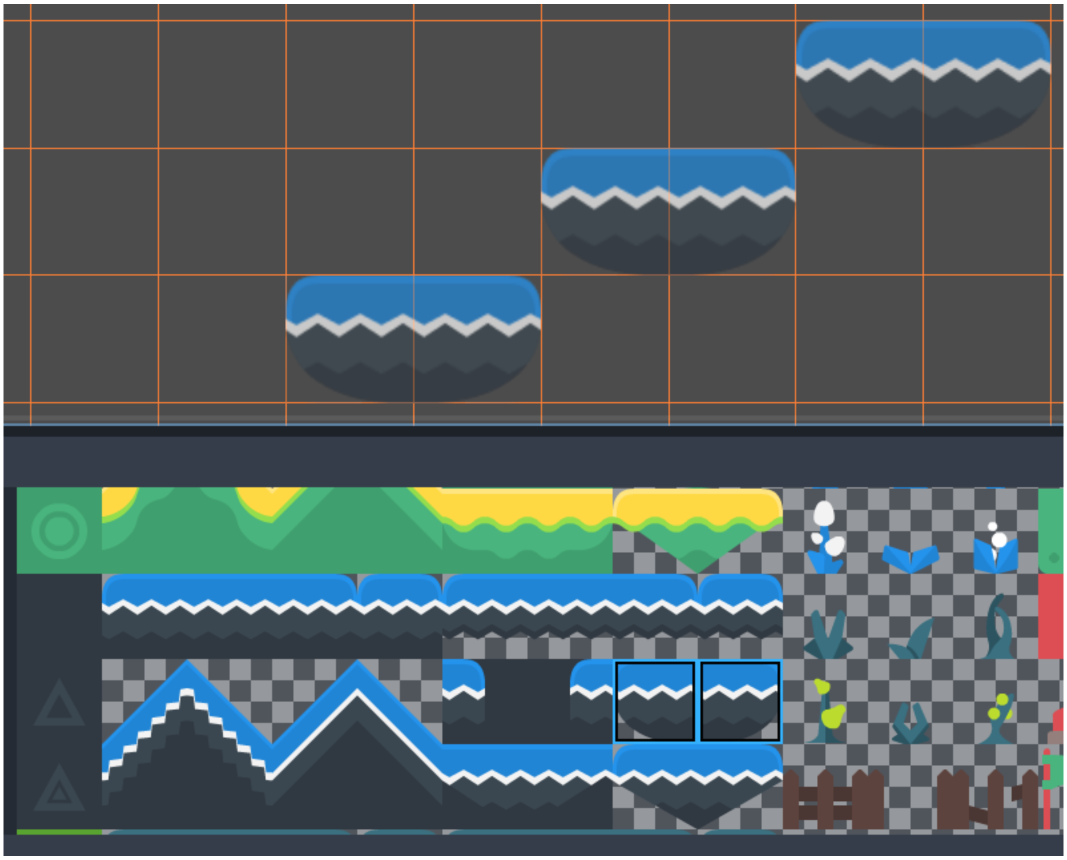
Line

After selecting Line Paint mode, you can draw in a line that is always 1 tile thick (no matter its orientation).

If you right-click while in Line Paint mode, you will erase in a line.

If you have selected multiple tiles in the TileMap or using the Selection tool, you can place them in a repeating pattern across the line.

You can toggle this mode temporarily while in Paint or Eraser mode by holding `Shift` then drawing.



Using the line tool after selecting two tiles to draw platforms diagonally

Rectangle

After selecting Rectangle Paint mode, you can draw in an axis-aligned rectangle.

If you right-click while in Rectangle Paint mode, you will erase in an axis-aligned rectangle.

If you have selected multiple tiles in the TileMap or using the Selection tool, you can place them in a repeating pattern within the rectangle.

You can toggle this mode temporarily while in Paint or Eraser mode by holding `Ctrl` and `Shift` then drawing.

Bucket Fill

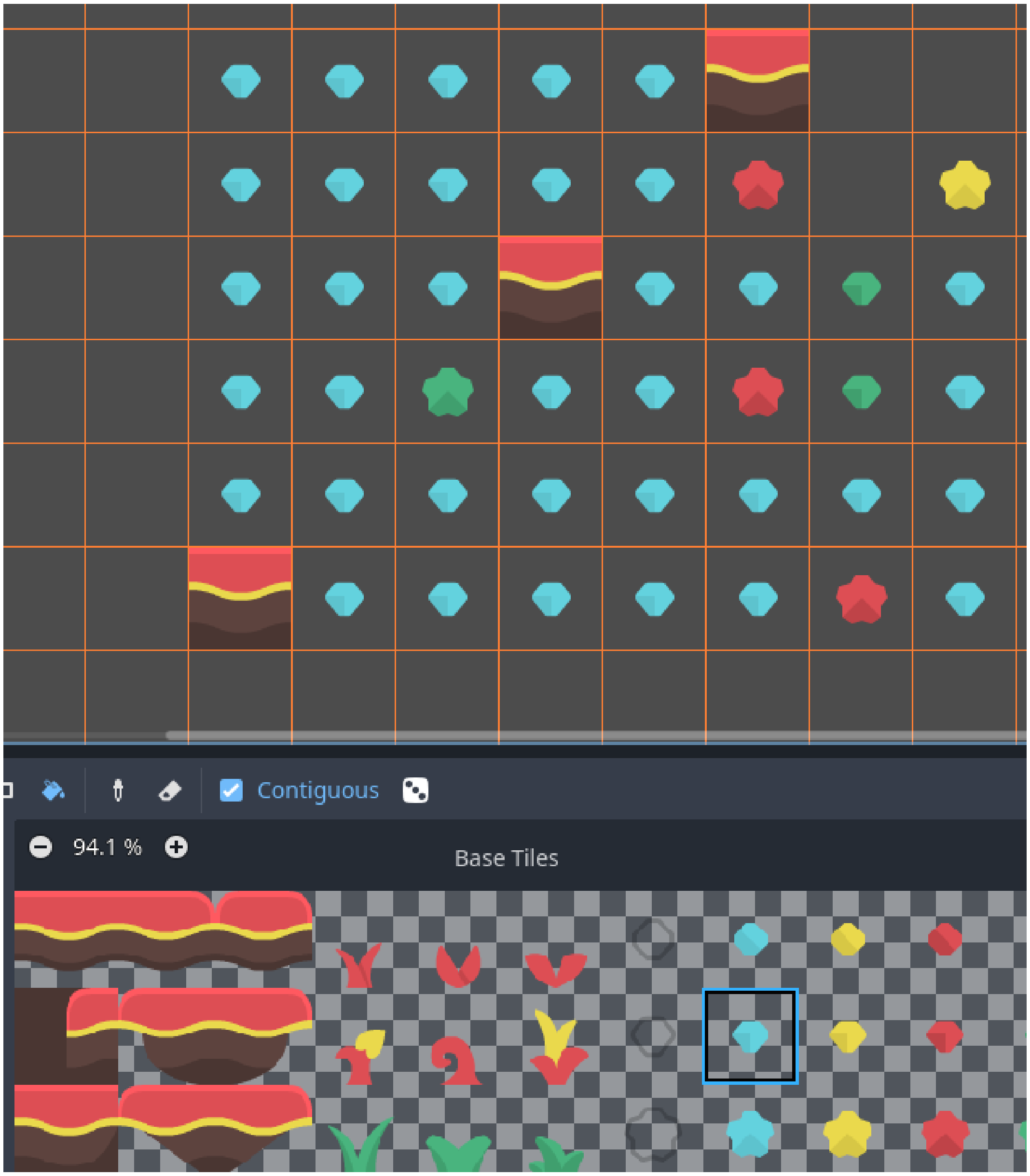
After selecting Bucket Fill mode, you can choose whether painting should be limited to contiguous areas only by toggling the **Contiguous** checkbox that appears on the right of the toolbar.

If you enable **Contiguous** (the default), only matching tiles that touch the current selection will be replaced. This contiguous check is performed horizontally and vertically, but *not* diagonally.

If you disable **Contiguous**, all tiles with the same ID in the entire TileMap will be replaced by the currently selected tile. If selecting an empty tile with **Contiguous** unchecked, all tiles in the rectangle that encompasses the TileMap's effective area will be replaced instead.

If you right-click while in Bucket Fill mode, you will replace matching tiles with empty tiles.

If you have selected multiple tiles in the TileMap or using the Selection tool, you can place them in a repeating pattern within the filled area.



Using the Bucket Fill tool

Picker

After selecting Picker mode, you can pick existing tiles in the 2D editor by holding `Ctrl` then clicking on a tile. This will switch the currently painted tile to the tile you've just clicked. You can also pick multiple tiles at once by holding down the left mouse button and forming a rectangle selection. Only non-empty tiles can be picked.

You can toggle this mode temporarily while in Paint mode by holding `Ctrl` then clicking or dragging the mouse.

Eraser

This mode is combined with any other painting mode (Paint, Line, Rectangle, Bucket Fill). When eraser mode is enabled, tiles will be replaced by empty tiles instead of drawing new lines when left-clicking.

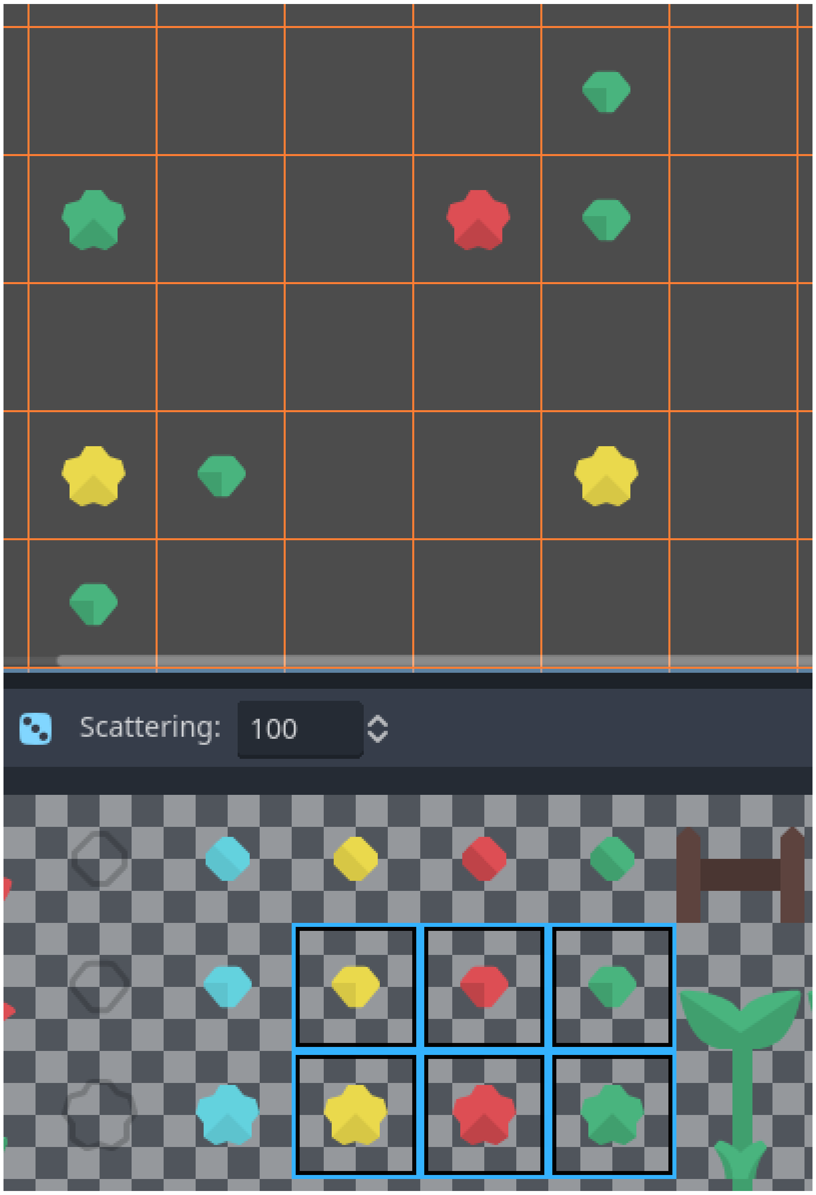
You can toggle this mode temporarily while in any other mode by right-clicking instead of left-clicking.

Painting randomly using scattering

While painting, you can optionally enable *randomization*. When enabled, a random tile will be chosen between all the currently selected tiles when painting. This is supported with the Paint, Line, Rectangle and Bucket Fill tools. For effective paint randomization, you must select multiple tiles in the TileMap editor or use scattering (both approaches can be combined).

If **Scattering** is set to a value greater than 0, there is a chance that no tile will be placed when painting. This can be used to add occasional, non-repeating detail to large areas (such as adding grass or crumbs on a large top-down TileMap).

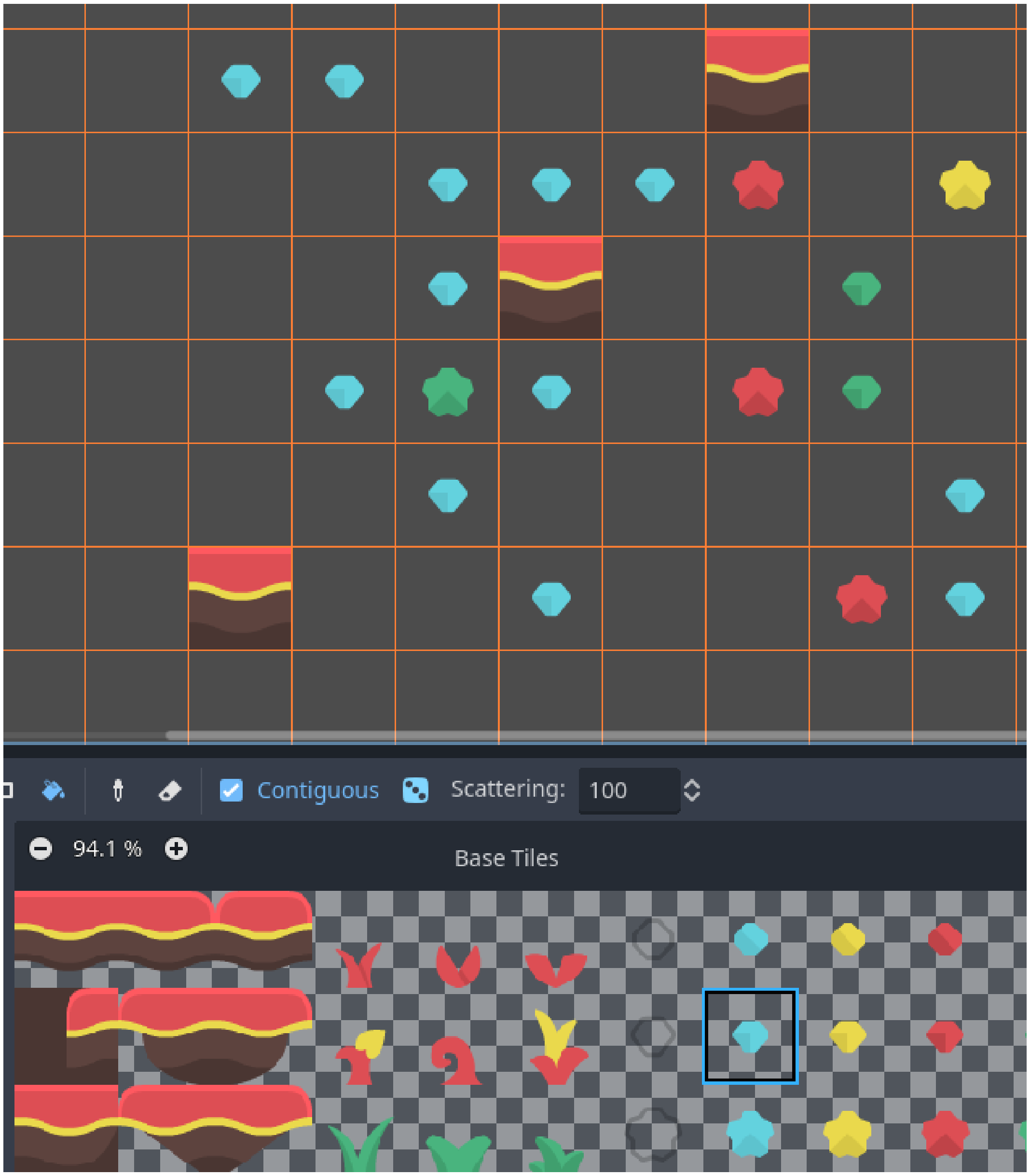
Example when using Paint mode:



Scattering: 100

Selecting from several times to randomly choose, then painting by holding down the left mouse button

Example when using Bucket Fill mode:


Contiguous
Scattering:
100
94.1 %
Base Tiles

Using Bucket Fill tool with a single tile, but with randomization and scattering enabled

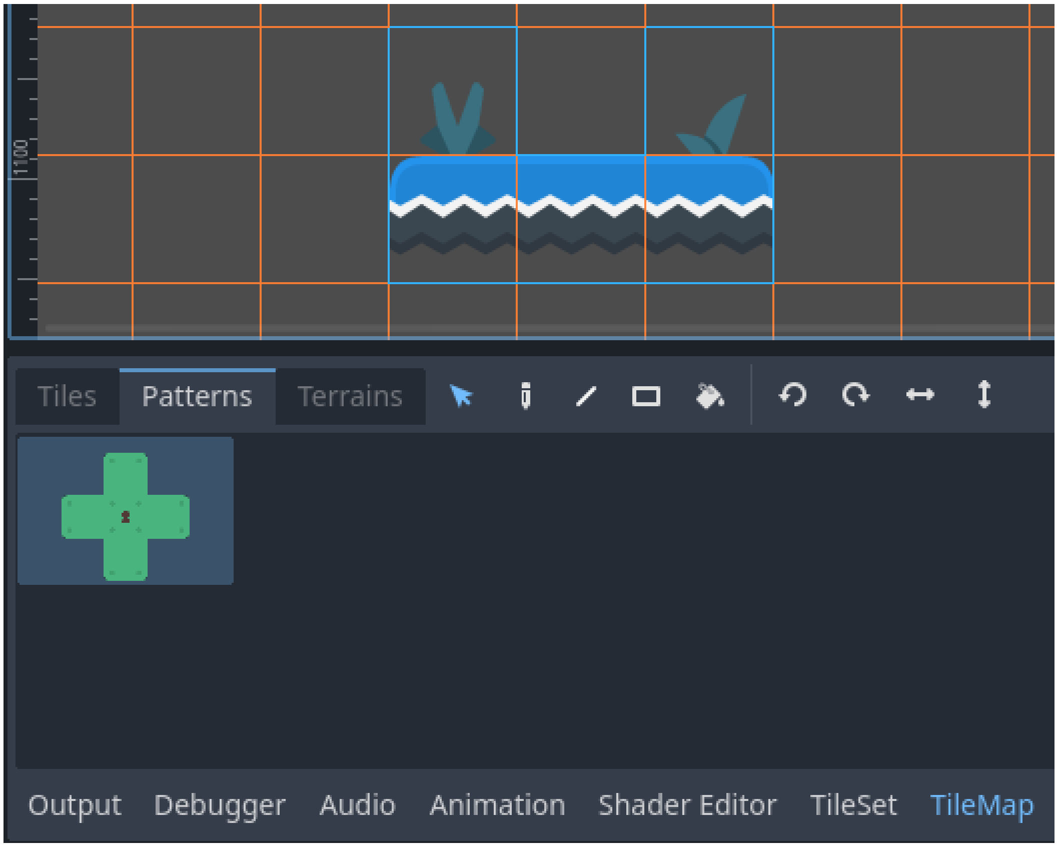
Note

Eraser mode does not take randomization and scattering into account. All tiles within the selection are always removed.

Saving and loading premade tile placements using patterns

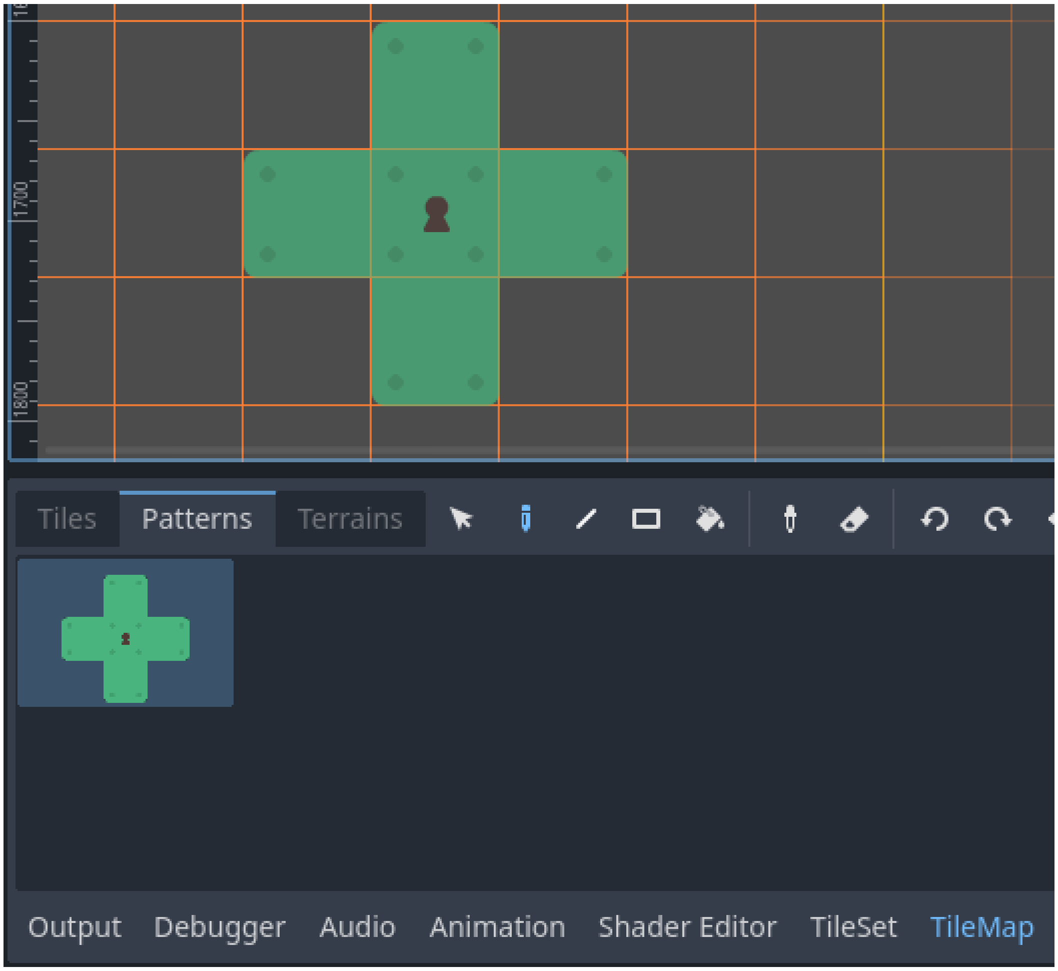
While you can copy and paste tiles while in Select mode, you may wish to save premade *patterns* of tiles to place together in a go. This can be done on a per-TileMap basis by choosing the **Patterns** tab of the TileMap editor.

To create a new pattern, switch to Select mode, perform a selection and press `Ctrl + C`. Click on empty space within the Patterns tab (a blue focus rectangle should appear around the empty space), then press `Ctrl + V`:


1100
Tiles
Patterns
Terrains
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Creating a new pattern from a selection in the TileMap editor

To use an existing pattern, click its image in the **Patterns** tab, switch to any painting mode, then left-click somewhere in the 2D editor:


1700
1800
Tiles
Patterns
Terrains
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Placing an existing pattern using the TileMap editor

Like multi-tile selections, patterns will be repeated if used with the Line, Rectangle or Bucket Fill painting modes.

Note

Despite being edited in the TileMap editor, patterns are stored in the TileSet resource. This allows reusing patterns in different

TileMapLayer nodes after loading a TileSet resource saved to an external file.

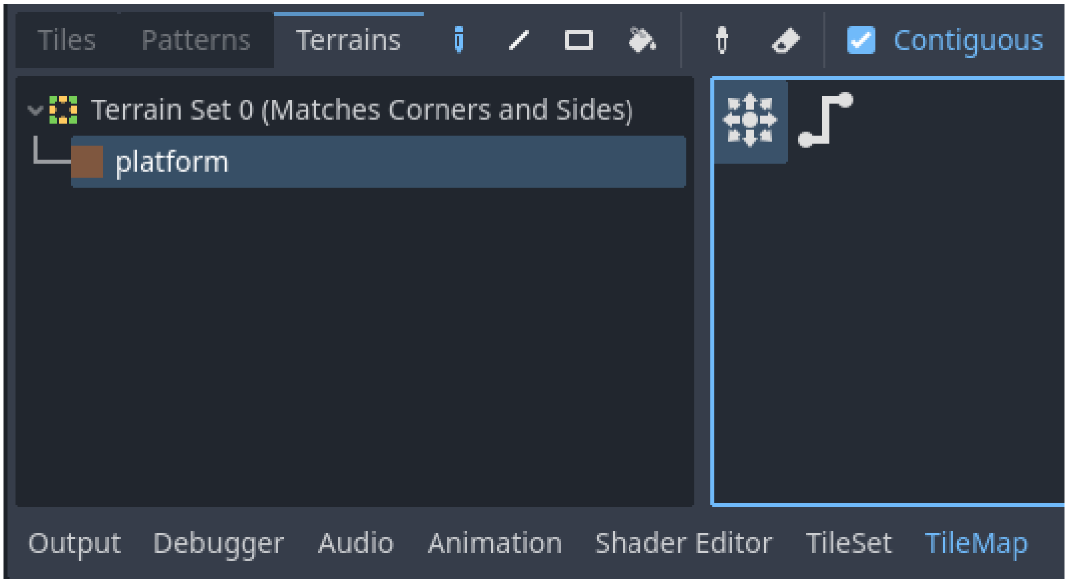
Handling tile connections automatically using terrains

To use terrains, the TileMapLayer node must feature at least one terrain set and a terrain within this terrain set. See Creating terrain sets (autotiling) if you haven't created a terrain set for the TileSet yet.

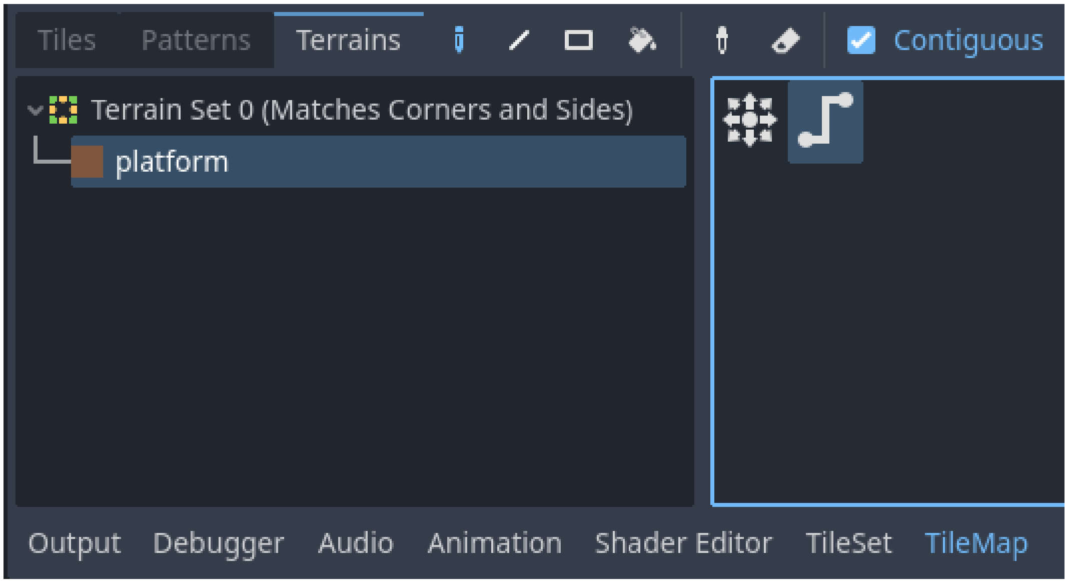
There are 3 kinds of painting modes available for terrain connections:

- **Connect**, where tiles are connected to surrounding tiles on the same TileMapLayer.
- **Path**, where tiles are connected to tiles painted in the same stroke (until the mouse button is released).
- Tile-specific overrides to resolve conflicts or handle situations not covered by the terrain system.

The Connect mode is easier to use, but Path is more flexible as it allows for more artist control during painting. For instance, Path can allow roads to be directly adjacent to each other without being connected to each other, while Connect will force both roads to be connected.

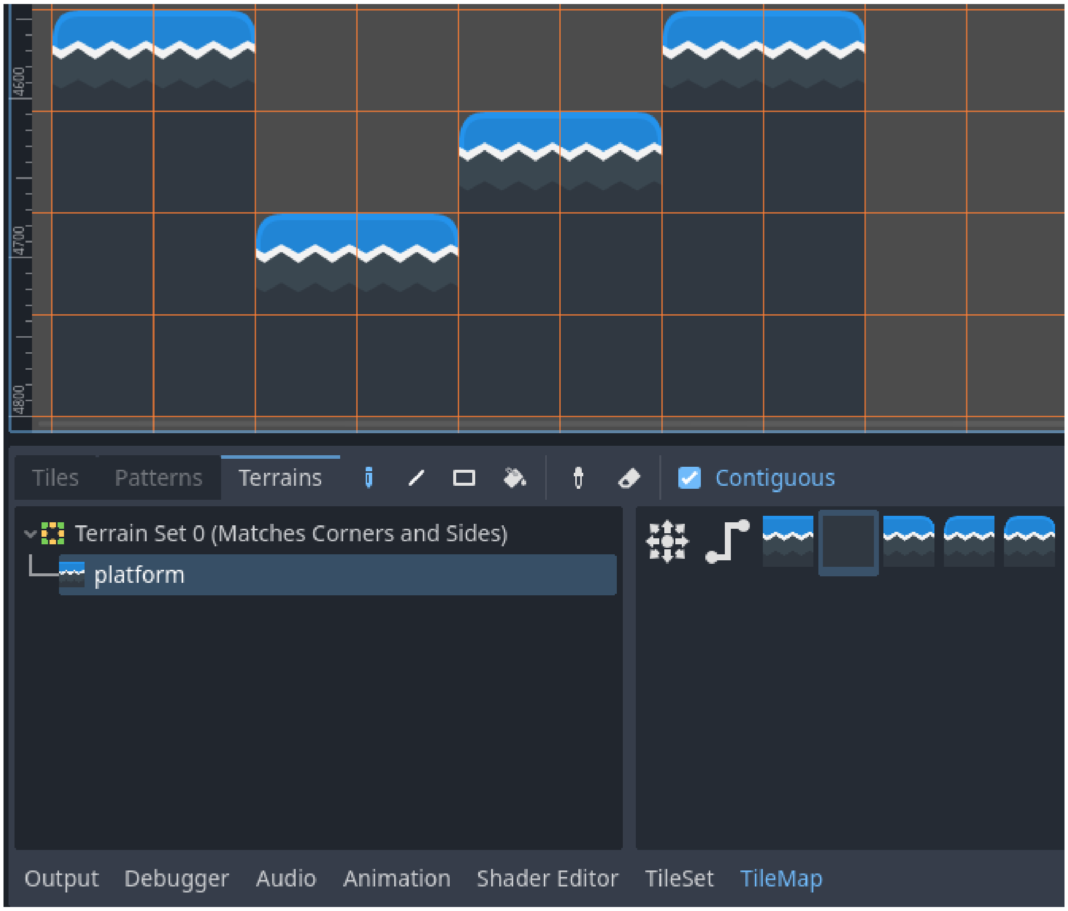

Tiles
Patterns
Terrains
Contiguous
Terrain Set 0 (Matches Corners and Sides)
platform
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Selecting Connect mode in the TileMap editor's Terrains tab


Tiles
Patterns
Terrains
Contiguous
Terrain Set 0 (Matches Corners and Sides)
platform
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Selecting Path mode in the TileMap editor's Terrains tab

Lastly, you can select specific tiles from the terrain to resolve conflicts in certain situations:

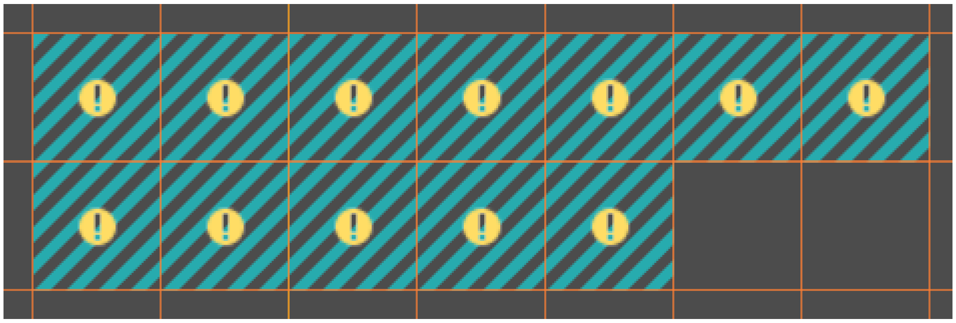

4600
4700
4800
Tiles
Patterns
Terrains
Contiguous
Terrain Set 0 (Matches Corners and Sides)
platform
Output
Debugger
Audio
Animation
Shader Editor
TileSet
TileMap

Painting with specific tiles in the TileMap editor's Terrains tab

Any tile that has at least one of its bits set to a value set to the corresponding terrain ID will appear in the list of tiles to choose from.

Handling missing tiles

If you remove tiles in the TileSet that are referenced in a TileMap, the TileMap will display a placeholder to indicate that an invalid tile ID is placed:



Missing tiles in the TileMap editor due to the TileSet reference being broken

These placeholders are **not** visible in the running project, but the tile data is still persisted to disk. This allows you to safely close and reopen such scenes. Once you re-add a tile with the matching ID, the tiles will appear with the new tile's appearance.

Note

Missing tile placeholders may not be visible until you select the TileMapLayer node and open the TileMap editor.

3D

- Introduction to 3D
- Using 3D transforms
- Procedural geometry
- 3D text

Rendering

- 3D rendering limitations
- Standard Material 3D and ORM Material 3D
- 3D lights and shadows
- Using decals
- Physical light and camera units
- Particle systems (3D)
- High dynamic range lighting
- Global illumination
- Environment and post-processing
- Volumetric fog and fog volumes
- 3D antialiasing

Optimization

- Using MultiMeshInstance3D
- Mesh level of detail (LOD)
- Visibility ranges (HLOD)
- Occlusion culling
- Resolution scaling
- Variable rate shading

Tools

- Prototyping levels with CSG
- Using GridMaps

Introduction to 3D

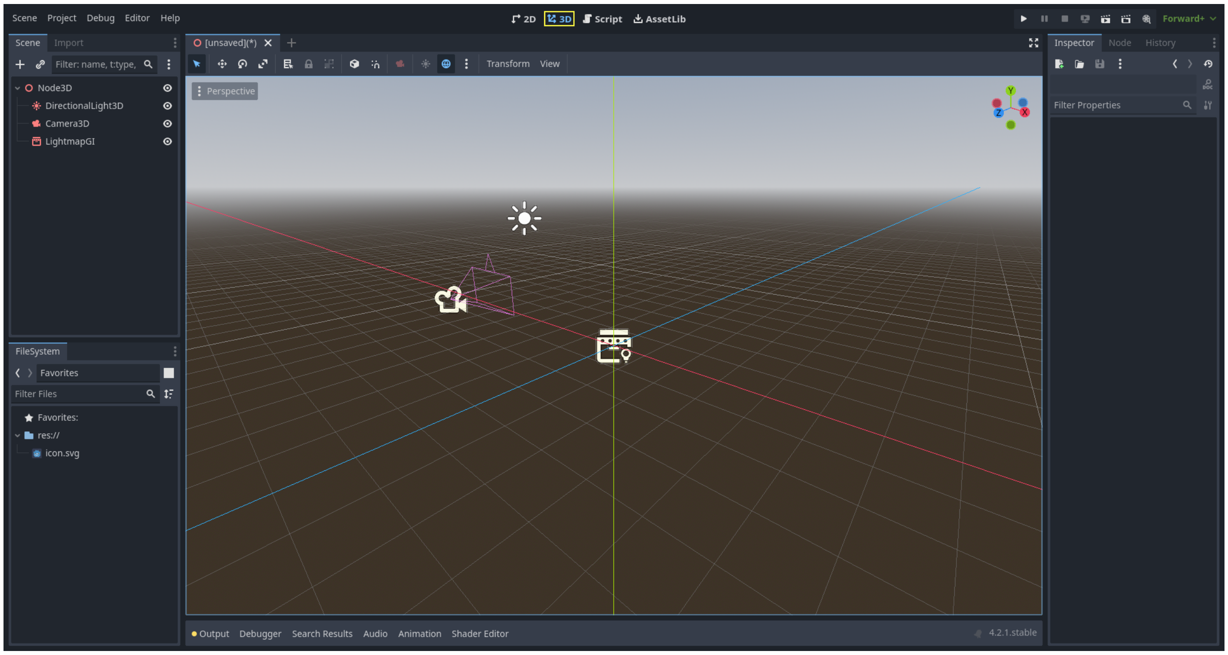
Creating a 3D game can be challenging. That extra Z coordinate makes many of the common techniques that helped to make 2D games simpler no longer work. To aid in this transition, it is worth mentioning that Godot uses similar APIs for 2D and 3D. Most nodes are the same and are present in both 2D and 3D versions. In fact, it is worth checking the 3D platformer tutorial, or the 3D kinematic character tutorials, which are almost identical to their 2D counterparts.

Godot Third Person Shooter (TPS) Demo, available on the Github repository [https://github.com/godotengine/tps-demo] or the Asset Library.

In 3D, math is a little more complex than in 2D, so also checking the Vector math entry in the wiki (which was especially created for game developers, not mathematicians or engineers) will help pave the way for you to develop 3D games efficiently.

3D workspace

Editing 3D scenes is done in the 3D workspace. This workspace can be selected manually, but it will be automatically selected when a Node3D node is selected.

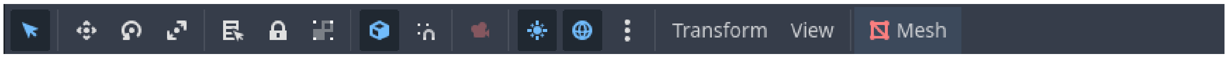

Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Forward+
Scene
Import
Filter: name, t:type,
Node3D
DirectionalLight3D
Camera3D
LightmapGI
[unsaved](*)
Transform
View
Perspective
Inspector
Node
History
Filter Properties
FileSystem
Favorites
Filter Files
Favorites:
res://
icon.svg
Output
Debugger
Search Results
Audio
Animation
Shader Editor
4.2.1.stable

Similar to 2D, the tabs below the workspace selector are used to change between currently opened scenes or create a new one using the plus (+) button. The left and right docks should be familiar from editor introduction.

Below the scene selector, the main toolbar is visible, and beneath the main toolbar is the 3D viewport.

Main toolbar

Some buttons in the main toolbar are the same as those in the 2D workspace. A brief explanation is given with the shortcut if the mouse cursor is hovered over a button for one second. Some buttons may have additional functionality if another keypress is performed. A recap of main functionality of each button with its default shortcut is provided below from left to right:



- **Select Mode** (`Q`): Allows selection of nodes in the viewport. Left clicking on a node to select one. Left clicking and dragging a rectangle selects all nodes within the rectangle's boundaries, once released. Holding `Shift` while selecting adds more nodes to the selection. Clicking on a selected node while holding `Shift` deselects the node. In this mode, you can use the gizmos to perform movement or rotation.

- **Move Mode** (`W`): Enables move (or translate) mode for the selected nodes. See Space and manipulation gizmos for more details.

- **Rotate Mode** (`E`): Enables rotation mode for the selected nodes. See Space and manipulation gizmos for more details.

- **Scale Mode** (`R`): Enables scaling and displays scaling gizmos in different axes for the selected nodes. See Space and manipulation gizmos for more details.

- **Show the list of selectable nodes at the clicked position**: As the description suggests, this provides a list of selectable nodes at the clicked position as a context menu, if there is more than one node in the clicked area.

- **Lock** (`Ctrl + L`) the selected nodes, preventing selection and movement in the viewport. Clicking the button again (or using `Ctrl + Shift + L`) unlocks the selected nodes. Locked nodes can only be selected in the scene tree. They can easily be identified with a padlock next to their node names in the scene tree. Clicking on this padlock also unlocks the nodes.

- **Group selected nodes** (`Ctrl + G`). This allows selection of the root node if any of the children are selected. Using `Ctrl + G` ungroups them. Additionally, clicking the ungroup button in the scene tree performs the same action.

- **Use Local Space** (`T`): If enabled, gizmos of a node are drawn using the current node's rotation angle instead of the global viewport axes.

- **Use Snap** (`Y`): If enabled, movement, and rotation snap to grid. Snapping can also temporarily be activated using `Ctrl` while performing the action. The settings for changing snap options are explained below.

- **Project Camera Override**: This action temporarily replaces the active camera in the level (e.g., the camera following the player) with the camera in the editor's viewport, allowing you to move freely and inspect the level's different parts, while game is running.

- **Toggle preview sunlight**: If no DirectionalLight3D exist in the scene, a preview of sunlight can be used as a light source. See Preview environment and light for more details.

- **Toggle preview environment**: If no WorldEnvironment exists in the scene, a preview of the environment can be used as a placeholder. See Preview environment and light for more details.

- **Edit Sun and Environment Settings (three dots)**: Opens the menu to configure preview sunlight and environment settings. See Preview environment and light for more details.

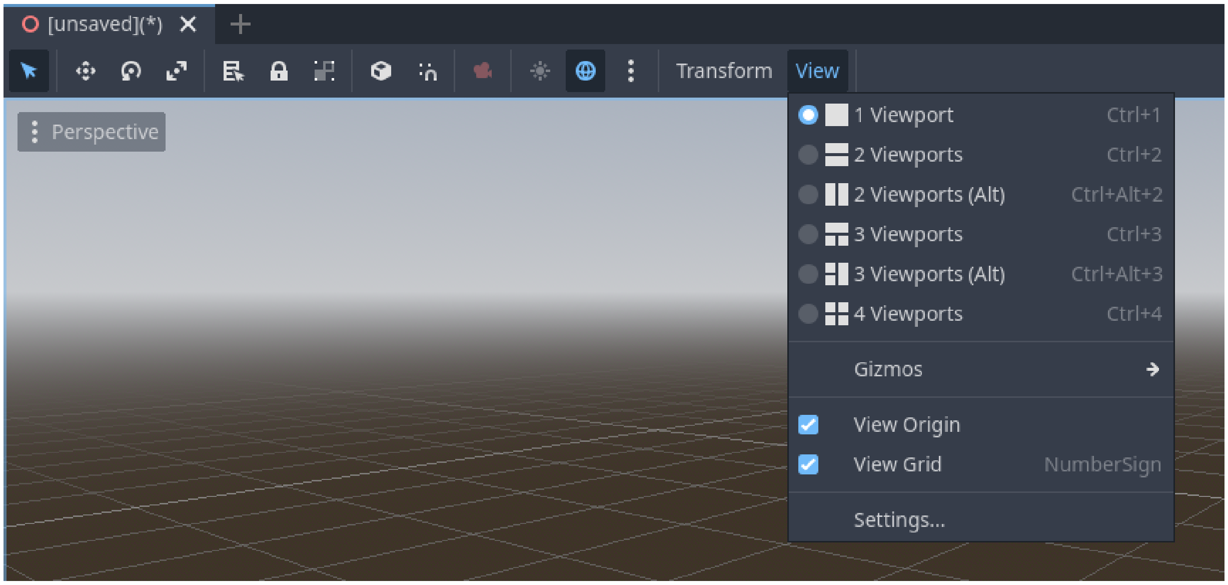
- **Transform menu**: It has three options:

    - *Snap Object to Floor*: Snaps an object to a solid floor.
    - *Transform Dialog*: Opens a dialog to adjust transform parameters (translate, rotate, scale, and transform)

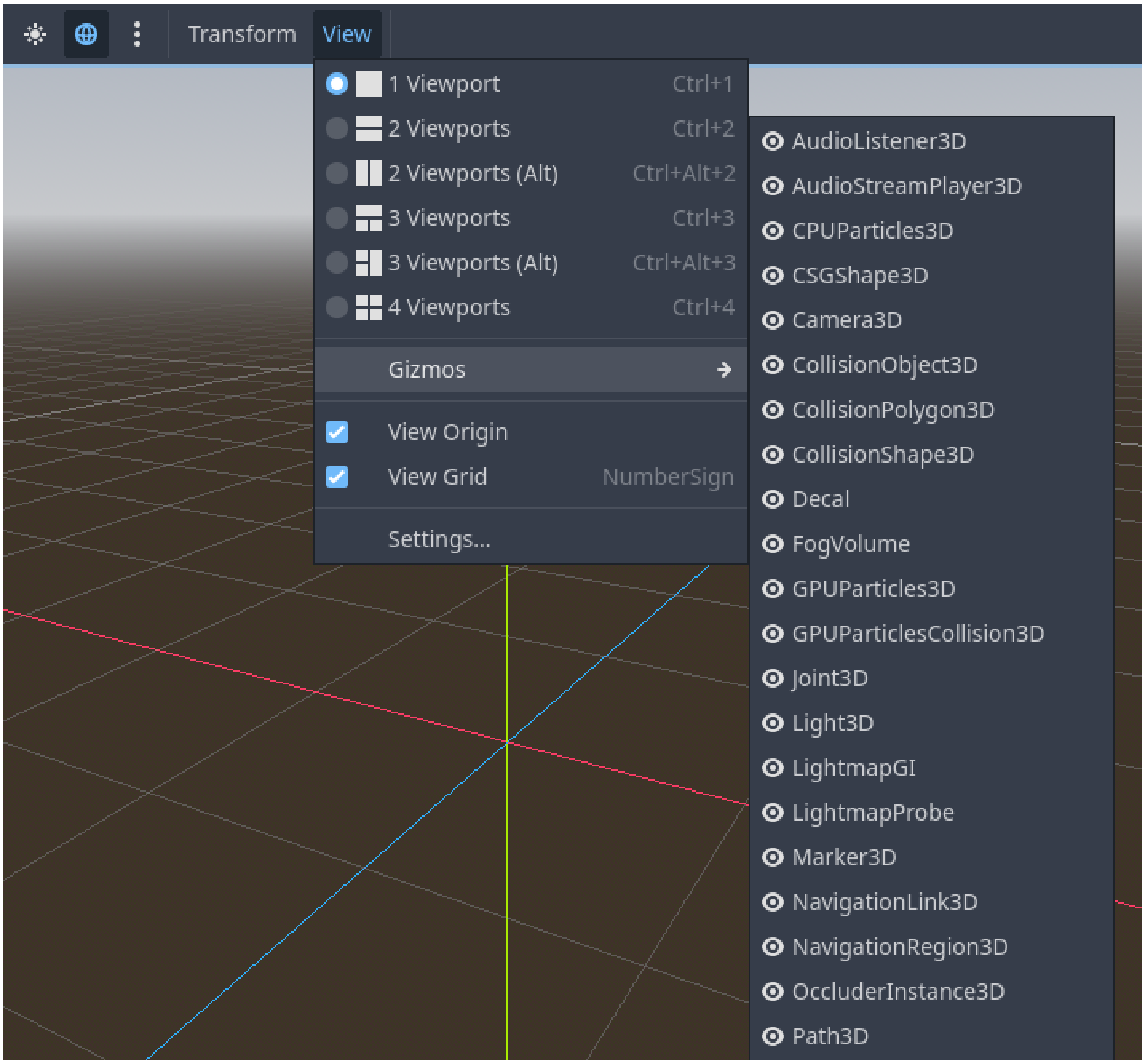
manually.

  - *Snap Settings*: Allows you to change transform, rotate snap (in degrees), and scale snap (in percent) settings.

- **View menu**: Controls the view options and enables additional viewports:


[unsaved](*)
Transform
View
Perspective
1 Viewport Ctrl+1
2 Viewports Ctrl+2
2 Viewports (Alt) Ctrl+Alt+2
3 Viewports Ctrl+3
3 Viewports (Alt) Ctrl+Alt+3
4 Viewports Ctrl+4
Gizmos
View Origin
View Grid NumberSign
Settings...

In this menu, you can also show/hide grids, which are set to 1x1 meter by default, and the origin, where the blue, green, and red axis lines intersect. Moreover, specific types of gizmos can be toggled in this menu.


Transform
View
1 Viewport Ctrl+1
2 Viewports Ctrl+2
2 Viewports (Alt) Ctrl+Alt+2
3 Viewports Ctrl+3
3 Viewports (Alt) Ctrl+Alt+3
4 Viewports Ctrl+4
Gizmos
View Origin
View Grid NumberSign
Settings...
AudioListener3D
AudioStreamPlayer3D
CPUParticles3D
CSGShape3D
Camera3D
CollisionObject3D
CollisionPolygon3D
CollisionShape3D
Decal
FogVolume
GPUParticles3D
GPUParticlesCollision3D
Joint3D
Light3D
LightmapGI
LightmapProbe
Marker3D
NavigationLink3D
NavigationRegion3D
OccluderInstance3D
Path3D

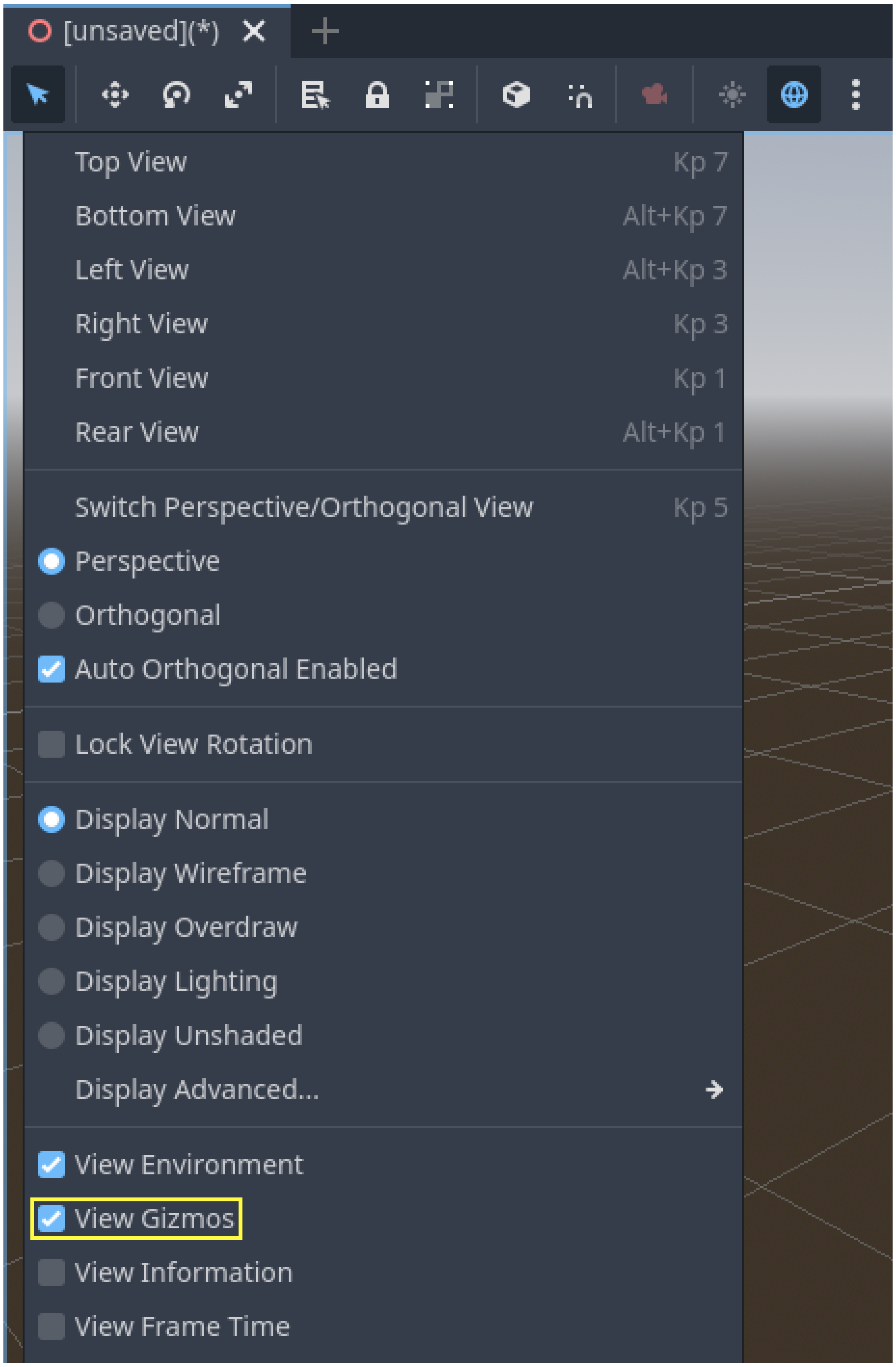
An open eye means that the gizmo is visible, a closed eye means it is hidden. A half-open eye means that it is also visible through opaque surfaces.

Clicking on *Settings* in this view menu opens a window to change the *Vertical Field of View (VFOV)* parameter (in degrees), *Z-Near*, and *Z-Far* values.

Next to the View menu, additional buttons may be visible. In the toolbar image at the beginning of this chapter, an additional *Mesh* button appears because a MeshInstance3D is selected. This menu provides some quick actions or tools to work on a specific node or selection.

View menu of viewport

Below the *Select* tool, in the 3D viewport, clicking on the three dots opens the **View menu** for the viewport. Hiding all shown gizmos in the editor's 3D view can also be performed through this menu:


[unsaved](*)
Top View Kp 7
Bottom View Alt+Kp 7
Left View Alt+Kp 3
Right View Kp 3
Front View Kp 1
Rear View Alt+Kp 1
Switch Perspective/Orthogonal View Kp 5
● Perspective
○ Orthogonal
☑ Auto Orthogonal Enabled
☐ Lock View Rotation
● Display Normal
○ Display Wireframe
○ Display Overdraw
○ Display Lighting
○ Display Unshaded
Display Advanced...
☑ View Environment
☑ View Gizmos
☐ View Information
☐ View Frame Time

This menu also displays the current view type and enables quick adjustment of the viewport's viewing angle. Additionally, it offers options to modify the appearance of nodes within the viewport.

Coordinate system

Godot uses the metric [https://en.wikipedia.org/wiki/Metric_system] system for everything in 3D, with 1 unit being equal to 1 meter. Physics and other areas are tuned for this scale. Therefore, attempting to use a different scale is usually a bad idea (unless you know what you are doing).

When working with 3D assets, it's always best to work in the correct scale (set the unit to metric in your 3D modeling software). Godot allows scaling post-import and, while this works in most cases, in rare situations it may introduce floating-point precision issues (and thus, glitches or artifacts) in delicate areas such as rendering or physics. Make sure your artists always work in the right scale!

The Y coordinate is used for "up". As for the horizontal X/Z axes, Godot uses a **right-handed** coordinate system. This means that for most objects that need alignment (such as lights or cameras), the Z axis is used as a "pointing towards" direction. This convention roughly means that:

- **X** is sides
- **Y** is up/down
- **Z** is front/back

See this chart for comparison with other 3D software:

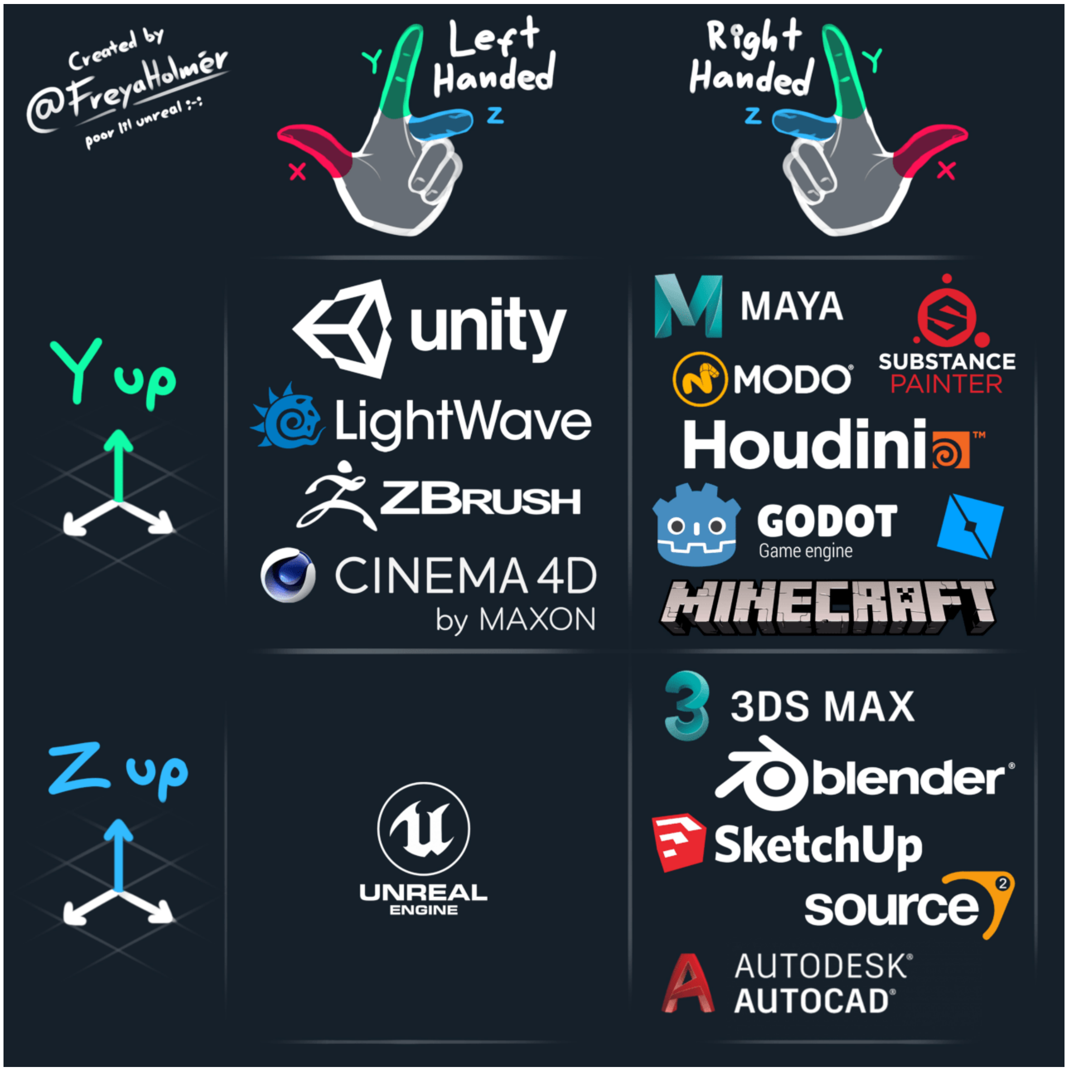
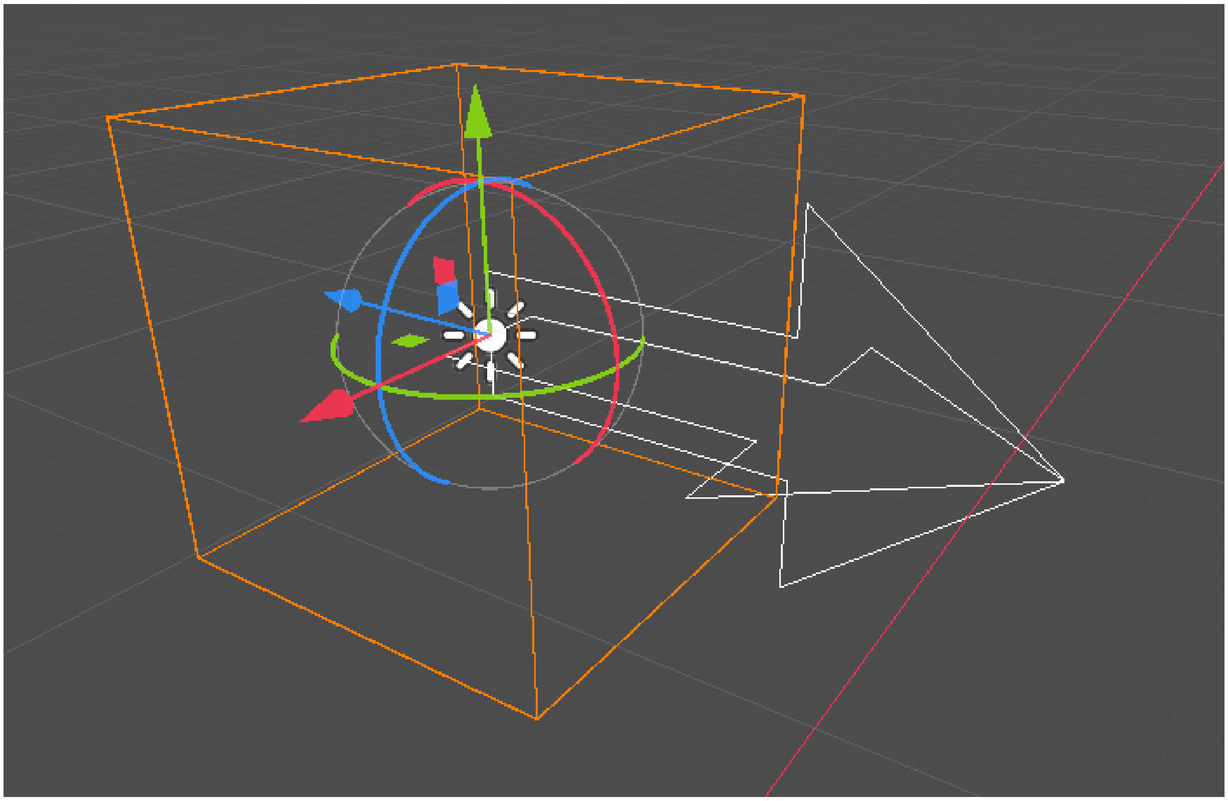

Created by
@FreyaHolmér
poor lil unreal :-:
Left Handed
Right Handed
Y
Z
X
Y up
Z up
unity
LightWave
ZBRUSH
CINEMA 4D
by MAXON
UNREAL ENGINE
MAYA
SUBSTANCE PAINTER
MODO
Houdini
GODOT
Game engine
MINECRAFT
3DS MAX
blender
SketchUp
source
AUTODESK
AUTOCAD

Image by [Freya Holmér](https://twitter.com/FreyaHolmer) [https://twitter.com/FreyaHolmer]

Space and manipulation gizmos

Moving, rotating, and scaling objects in the 3D view is done through the manipulator gizmos. Each axis is represented by a color: Red, Green, Blue represent X, Y, Z respectively. This convention applies to the grid and other gizmos too (and also to the shader language, ordering of components for Vector3, Color, etc.).



Some useful keybindings:

- To snap placement or rotation, press `Ctrl` while moving, scaling, or rotating.
- To center the view on the selected object, press `F`.

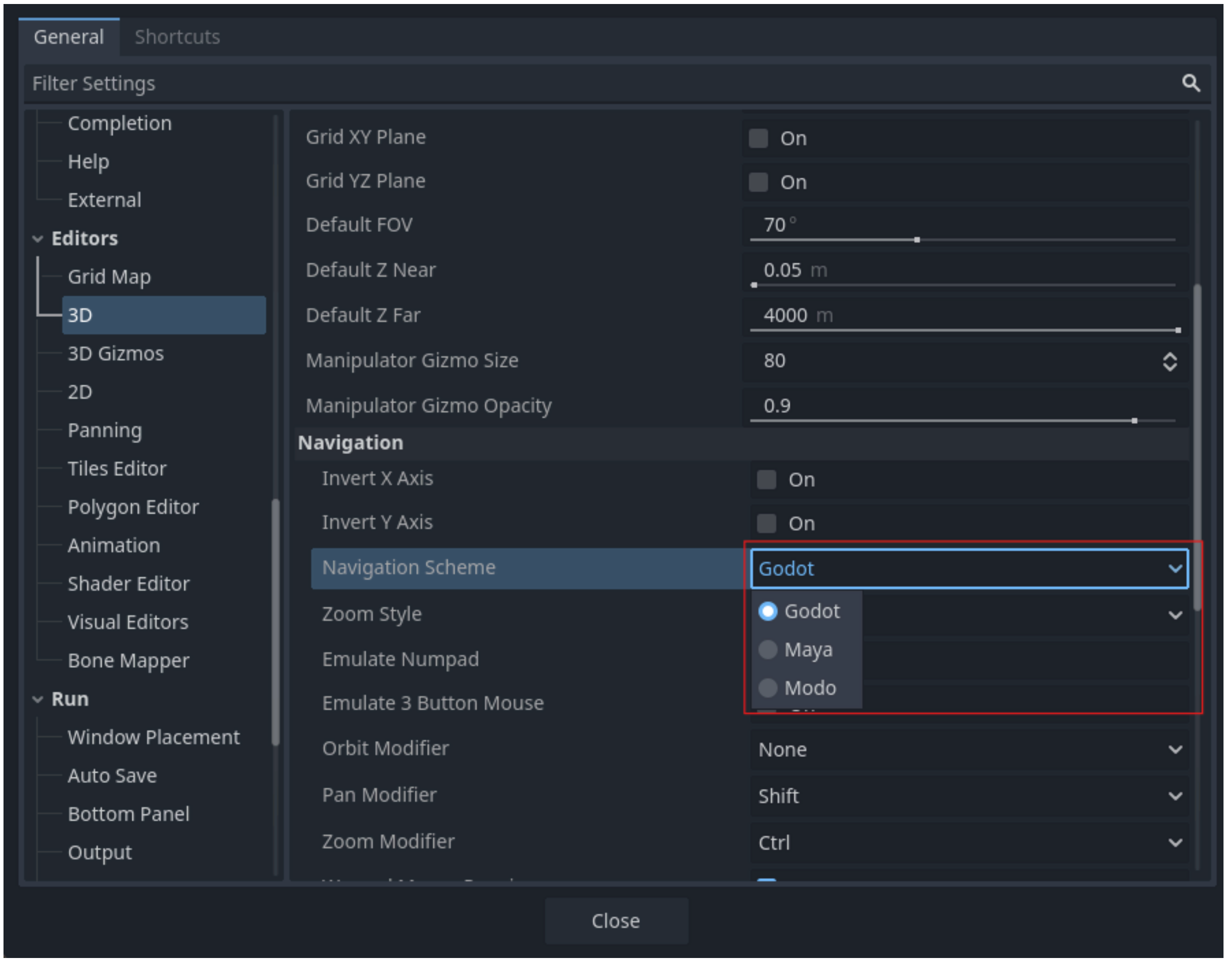
In the viewport, the arrows can be clicked and held to move the object on an axis. The arcs can be clicked and held to rotate the object. To lock one axis and move the object freely in the other two axes, the colored rectangles can be clicked, held, and dragged.

If the transform mode is changed from *Select Mode* to *Scale Mode*, the arrows will be replaced by cubes, which can be dragged to scale an object as if the object is being moved.

Navigating the 3D environment

In 3D environments, it is often important to adjust the viewpoint or angle from which you are viewing the scene. In Godot, navigating the 3D environment in the viewport (or spatial editor) can be done in multiple ways.

The default 3D scene navigation controls are similar to Blender (aiming to have some sort of consistency in the free software pipeline), but options are included to customize mouse buttons and behavior to be similar to other tools in the Editor Settings. To change the controls to Maya or Modo controls, you can navigate to **Editor Settings > Editors > 3D**. Then, under *Navigation*, search for *Navigation Scheme*.


General
Shortcuts
Filter Settings
Completion
Help
External
Editors
Grid Map
3D
3D Gizmos
2D
Panning
Tiles Editor
Polygon Editor
Animation
Shader Editor
Visual Editors
Bone Mapper
Run
Window Placement
Auto Save
Bottom Panel
Output
Close

Grid XY Plane ☐ On
Grid YZ Plane ☐ On
Default FOV 70°
Default Z Near 0.05 m
Default Z Far 4000 m
Manipulator Gizmo Size 80
Manipulator Gizmo Opacity 0.9

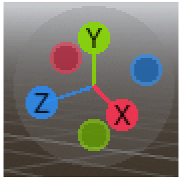
Navigation

Invert X Axis ☐ On
Invert Y Axis ☐ On
Navigation Scheme Godot
● Godot
○ Maya
○ Modo
Zoom Style
Emulate Numpad
Emulate 3 Button Mouse
Orbit Modifier None
Pan Modifier Shift
Zoom Modifier Ctrl

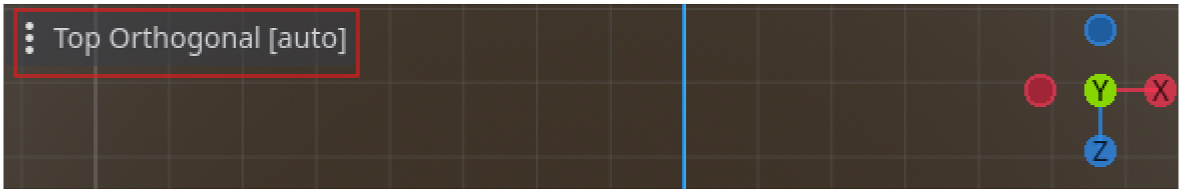
Using the default settings, the following shortcuts control how one can navigate in the viewport:

Pressing the middle mouse button and dragging the mouse allows you to orbit around the center of what is on the screen.

It is also possible to left-click and hold the manipulator gizmo located on the top right of the viewport to orbit around the center:



Left-clicking on one of the colored circles will set the view to the chosen orthogonal and the viewport's view menu will be updated accordingly.



If the *Perspective* view is enabled on the viewport (can be seen on the viewport's View menu, not the View menu on the main toolbar), holding down the right mouse button on the viewport or pressing `Shift + F` switches to "free-look" mode. In this mode you can move the mouse to look around, use the `W A S D` keys to fly around the view, `E` to go up, and `Q` to go down. To disable this mode, release the right mouse button or press `Shift + F` again.

In the free-look mode, you can temporarily increase the flying speed using `Shift` or decrease it using `Alt`. To change and keep the speed modifier use `mouse wheel up` or `mouse wheel down`, to increase or decrease it, respectively.

In orthogonal mode, holding the right mouse button will pan the view instead. Use `Keypad 5` to toggle between perspective and orthogonal view.

Using Blender-style transform shortcuts

Since Godot 4.2, you can enable Blender-style shortcuts for translating, rotating and scaling nodes. In Blender, these shortcuts are:

- `G` for translating
- `R` for rotating
- `S` for scaling

After pressing a shortcut key while focusing on the 3D editor viewport, move the mouse or enter a number to move the selected node(s) by the specified amount in 3D units. You can constrain movement to a specific axis by specifying the axis as a letter, then the distance (if entering a value with the keyboard).

For instance, to move the selection upwards by 2.5 units, enter the following sequence in order (Y+ is upwards in Godot):

`G`-`Y`-`2`-`.`-`5`-`Enter`

To use Blender-style transform shortcuts in Godot, go to the Editor Settings' **Shortcuts** tab, then in the Spatial Editor section:

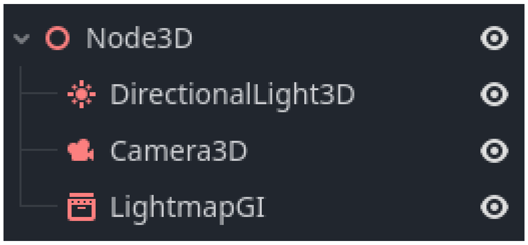
- Bind **Begin Translate Transformation** to `G`.
- Bind **Begin Rotate Transformation** to `R`.
- Bind **Begin Scale Transformation** to `S`.
- Finally, unbind **Scale Mode** so that its shortcut won't conflict with **Begin Rotate Transformation**.

Tip

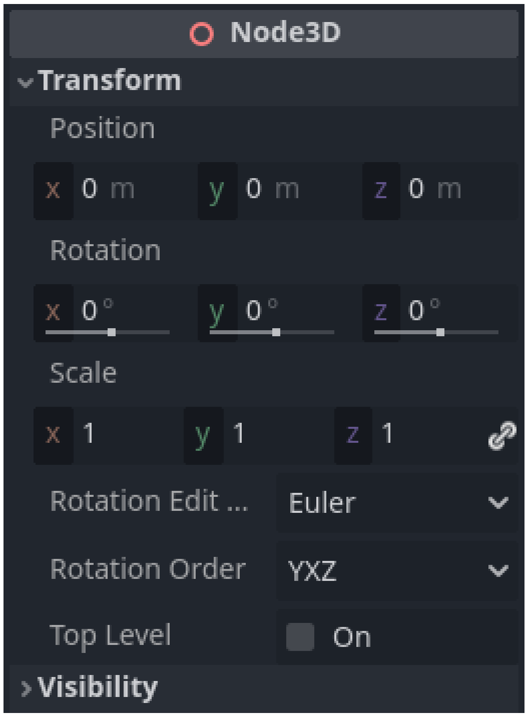
More shortcuts can be found on the 3D / Spatial Editor page.

Node3D node

Node2D is the base node for 2D. Control is the base node for everything GUI. Following this reasoning, the 3D engine uses the Node3D node for everything 3D.



Node3Ds have a local transform, which is relative to the parent node (as long as the parent node is also of **or inherits from** the type Node3D). This transform can be accessed as a 3×4 Transform3D, or as 3 Vector3 members representing location, Euler rotation (X, Y and Z angles) and scale.



Node3D

Transform

Position

x 0 m y 0 m z 0 m

Rotation

x 0° y 0° z 0°

Scale

x 1 y 1 z 1

Rotation Edit ... Euler

Rotation Order YXZ

Top Level ☐ On

Visibility

3D content

Unlike 2D, where loading image content and drawing is straightforward, 3D is a little more difficult. The content needs to be created with special 3D tools (also called Digital Content Creation tools, or DCCs) and exported to an exchange file format to be imported in Godot. This is required since 3D formats are not as standardized as images.

Manually authored models (using 3D modeling software)

It is possible to import 3D models in Godot created in external tools. Depending on the format, you can import entire scenes (exactly as they look in the 3D modeling software), including animation, skeletal rigs, blend shapes, or as simple resources.

See also

See Importing 3D scenes for more on importing.

Generated geometry

It is possible to create custom geometry by using the ArrayMesh resource directly. Simply create your arrays and use the ArrayMesh.add_surface_from_arrays() function. A helper class is also available, SurfaceTool, which provides a more straightforward API and helpers for indexing, generating normals, tangents, etc.

In any case, this method is meant for generating static geometry (models that will not be updated often), as creating vertex arrays and submitting them to the 3D API has a significant performance cost.

Note

To learn about prototyping inside Godot or using external tools, see Prototyping levels with CSG.

Immediate geometry

If, instead, you need to generate simple geometry that will be updated often, Godot provides a special ImmediateMesh resource that can be used in a MeshInstance3D node. This provides an OpenGL 1.x-style immediate-mode API to create points, lines, triangles, etc.

2D in 3D

While Godot packs a powerful 2D engine, many types of games use 2D in a 3D environment. By using a fixed camera (either orthogonal or perspective) that does not rotate, nodes such as Sprite3D and AnimatedSprite3D can be used to create 2D games that take advantage of mixing with 3D backgrounds, more realistic parallax, lighting/shadow effects, etc.

The disadvantage is, of course, that added complexity and reduced performance in comparison to plain 2D, as well as the lack of reference of working in pixels.

Environment

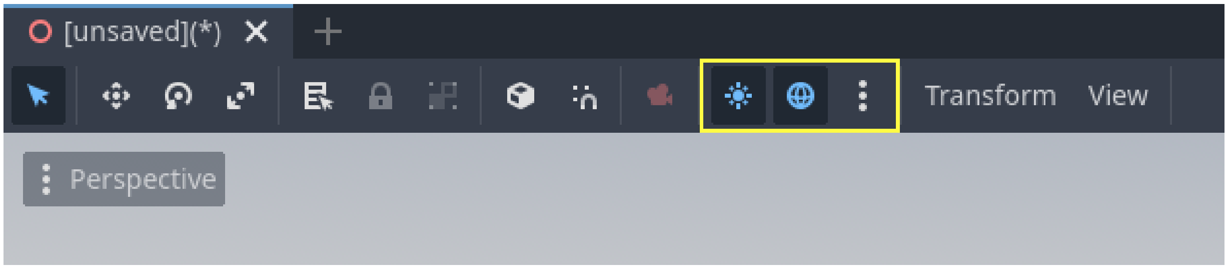
Besides editing a scene, it is often common to edit the environment. Godot provides a WorldEnvironment node that allows changing the background color, mode (as in, put a skybox), and applying several types of built-in post-processing effects. Environments can also be overridden in the Camera.

Preview environment and light

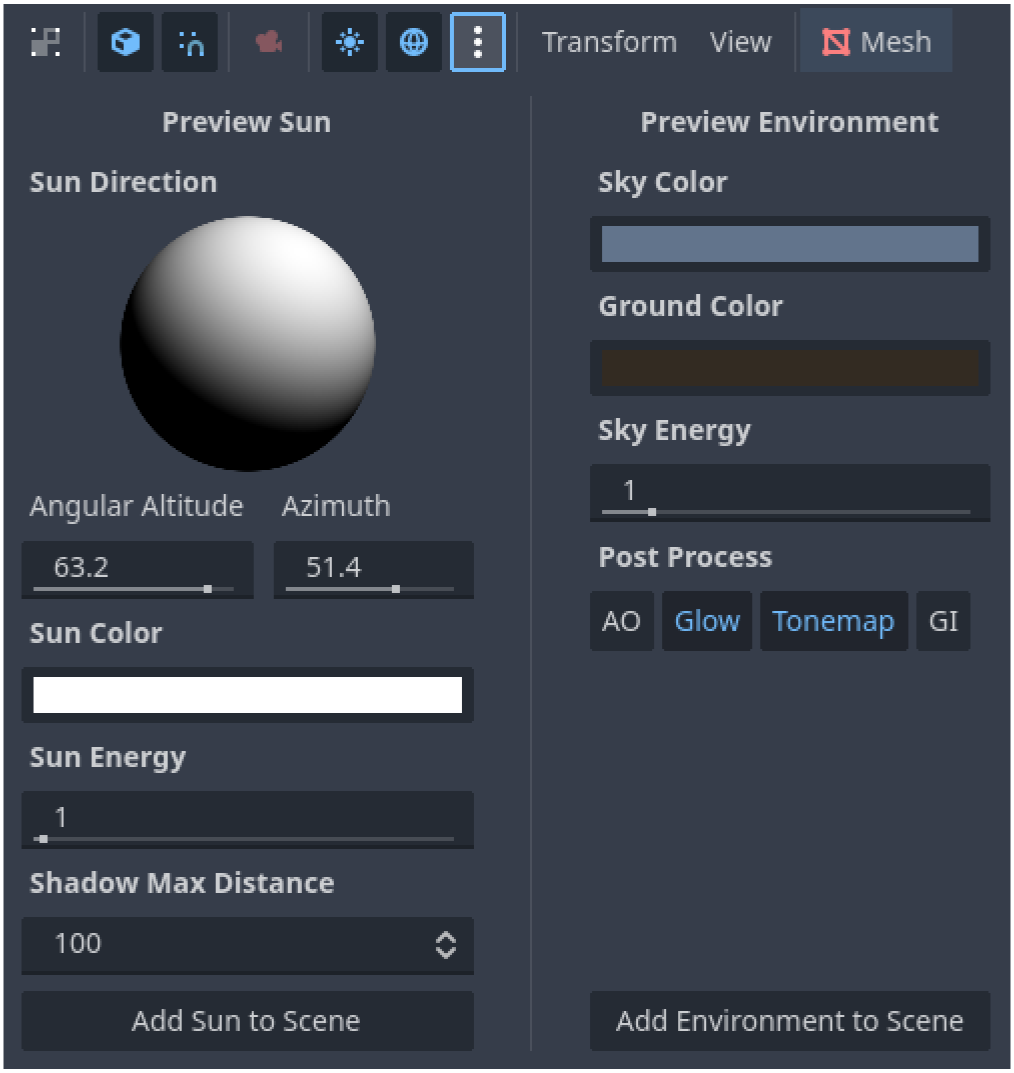
By default, any 3D scene that doesn't have a [WorldEnvironment]() node, or a [DirectionalLight3D](), will have a preview turned on for what it's missing to light the scene.

The preview light and environment will only be visible in the scene while in the editor. If you run the scene or export the project they will not affect the scene.

The preview light and environment can be turned on or off from the top menu by clicking on their respective icon.



The three dots dropdown menu next to those icons can be used to adjust the properties of the preview environment and light if they are enabled.


Transform
View
Mesh
AO
Glow
Tonemap
GI
Add Sun to Scene
Add Environment to Scene

Preview Sun

Sun Direction

Angular Altitude 63.2

Azimuth 51.4

Sun Color

Sun Energy 1

Shadow Max Distance 100

Preview Environment

Sky Color

Ground Color

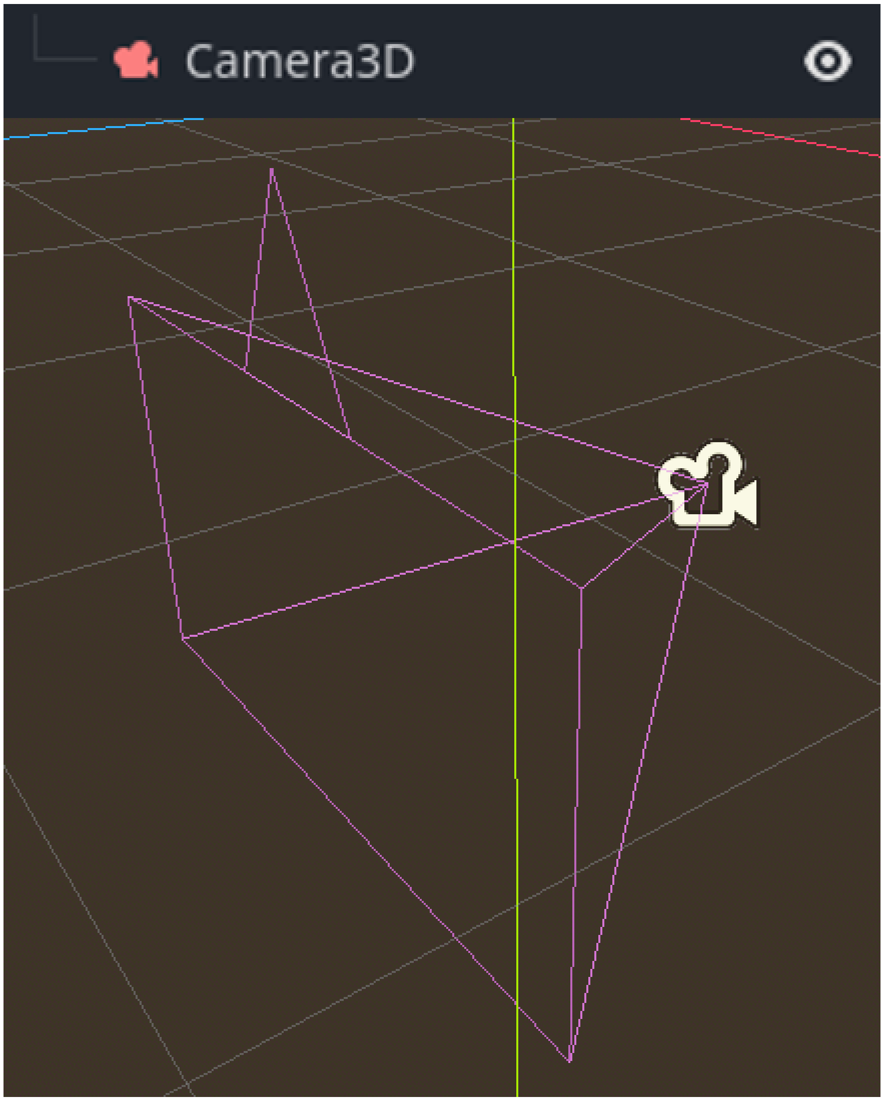
Sky Energy 1

Post Process

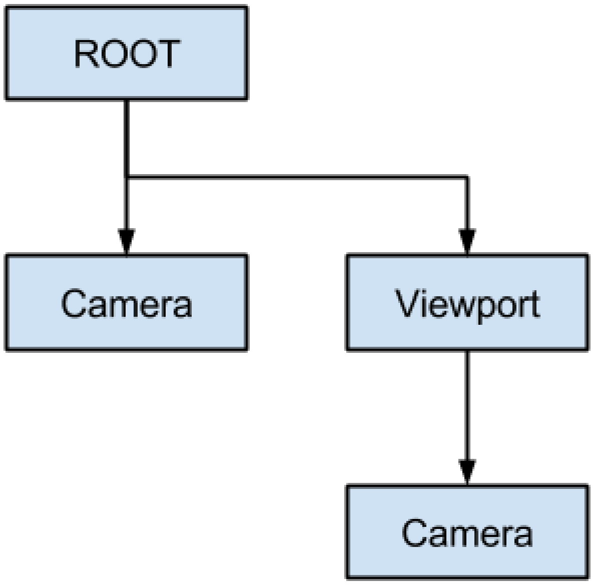
The same preview sun and environment is used for every scene in the same project, So only make adjustments that would apply to all of the scenes you will need a preview light and environment for.

Cameras

No matter how many objects are placed in the 3D space, nothing will be displayed unless a Camera3D is also added to the scene. Cameras can work in either orthogonal or perspective projections:



Cameras are associated with (and only display to) a parent or grandparent viewport. Since the root of the scene tree is a viewport, cameras will display on it by default, but if sub-viewports (either as render target or picture-in-picture) are desired, they need their own children cameras to display.



When dealing with multiple cameras, the following rules are enforced for each viewport:

- If no cameras are present in the scene tree, the first one that enters it will become the active camera. Further cameras entering the scene will be ignored (unless they are set as *current*).
- If a camera has the "*current*" property set, it will be used regardless of any other camera in the scene. If the property is set, it will become active, replacing the previous camera.
- If an active camera leaves the scene tree, the first camera in tree-order will take its place.

Lights

The background environment emits some ambient light which appears on surfaces. Still, without any light sources placed in the scene, the scene will appear quite dark unless the background environment is very bright.

Most outdoor scenes have a directional light (the sun or moon), while indoor scenes typically have several positional lights (lamps, torches, …). See 3D lights and shadows for more information on setting up lights in Godot.

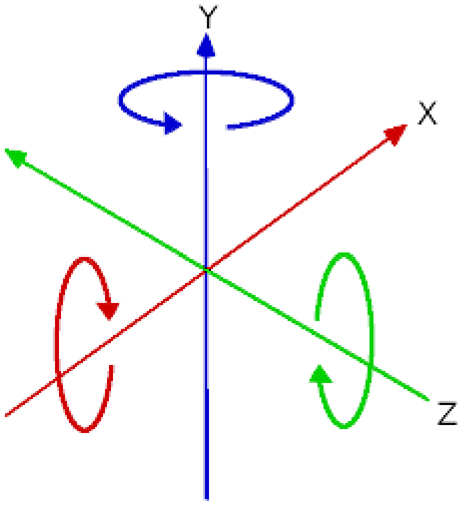
Using 3D transforms

Introduction

If you have never made 3D games before, working with rotations in three dimensions can be confusing at first. Coming from 2D, the natural way of thinking is along the lines of *"Oh, it's just like rotating in 2D, except now rotations happen in X, Y and Z"*.

At first, this seems easy. For simple games, this way of thinking may even be enough. Unfortunately, it's often incorrect.

Angles in three dimensions are most commonly referred to as "Euler Angles".



Euler angles were introduced by mathematician Leonhard Euler in the early 1700s.

This way of representing 3D rotations was groundbreaking at the time, but it has several shortcomings when used in game development (which is to be expected from a guy with a funny hat). The idea of this document is to explain why, as well as outlining best practices for dealing with transforms when programming 3D games.

Problems of Euler angles

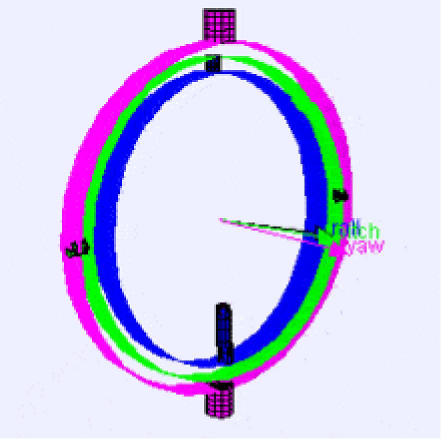
While it may seem intuitive that each axis has a rotation, the truth is that it's just not practical.

Axis order

The main reason for this is that there isn't a *unique* way to construct an orientation from the angles. There isn't a standard mathematical function that takes all the angles together and produces an actual 3D rotation. The only way an orientation can be produced from angles is to rotate the object angle by angle, in an *arbitrary order*.

This could be done by first rotating in *X*, then *Y* and then in *Z*. Alternatively, you could first rotate in *Y*, then in *Z* and finally in *X*. Anything works, but depending on the order, the final orientation of the object will *not necessarily be the same*. Indeed, this means that there are several ways to construct an orientation from 3 different angles, depending on *the order of the rotations*.

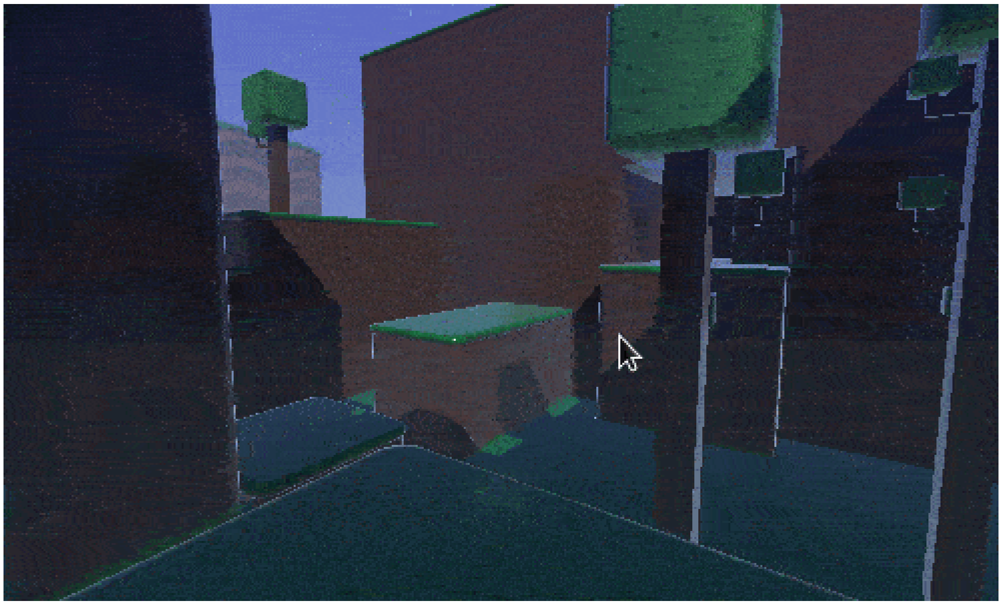
Following is a visualization of rotation axes (in X, Y, Z order) in a gimbal (from Wikipedia). As you can see, the orientation of each axis depends on the rotation of the previous one:



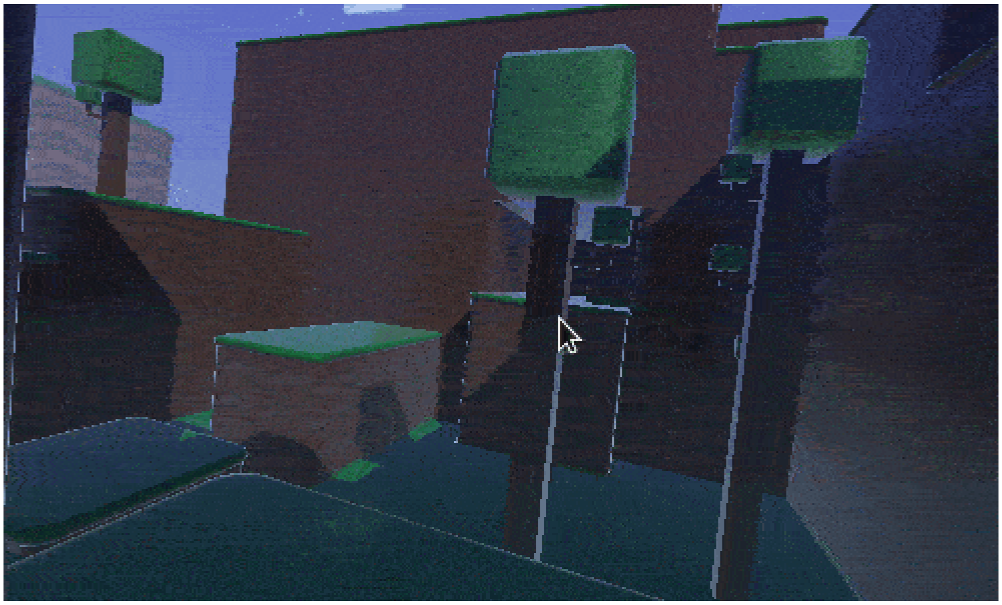
You may be wondering how this affects you. Let's look at a practical example:

Imagine you are working on a first-person controller (e.g. an FPS game). Moving the mouse left and right controls your view angle parallel to the ground, while moving it up and down moves the player's view up and down.

In this case to achieve the desired effect, rotation must be applied first in the *Y* axis ("up" in this case, since Godot uses a "Y-Up" orientation), followed by rotation in the *X* axis.



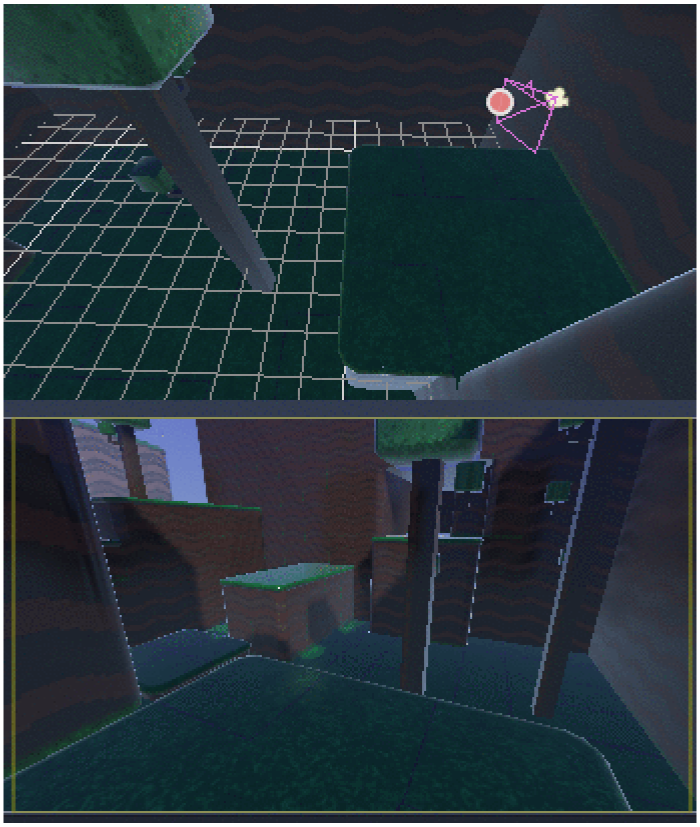
If we were to apply rotation in the *X* axis first, and then in *Y*, the effect would be undesired:



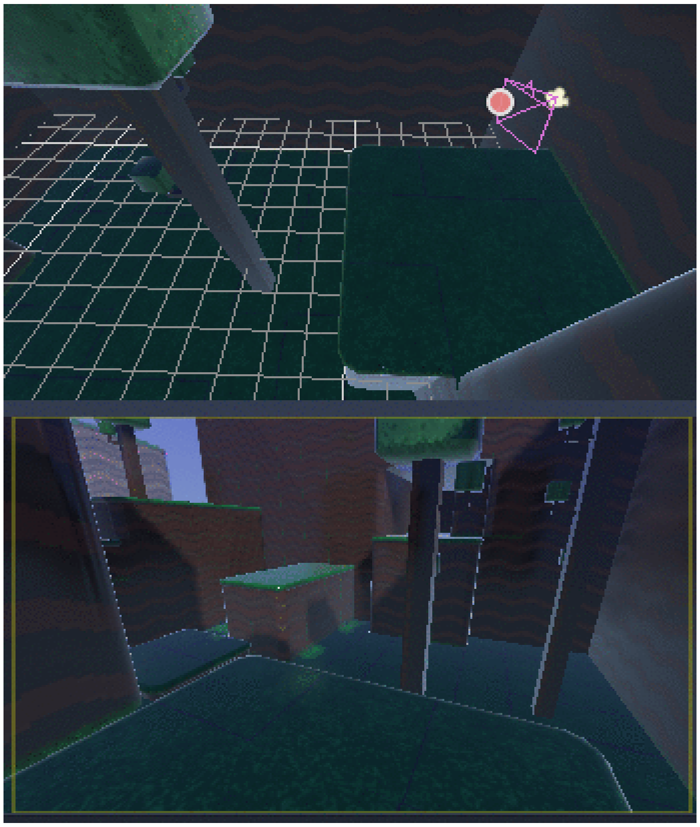
Depending on the type of game or effect desired, the order in which you want axis rotations to be applied may differ. Therefore, applying rotations in X, Y, and Z is not enough: you also need a *rotation order*.

Interpolation

Another problem with using Euler angles is interpolation. Imagine you want to transition between two different camera or enemy positions (including rotations). One logical way to approach this is to interpolate the angles from one position to the next. One would expect it to look like this:



But this does not always have the expected effect when using angles:



The camera actually rotated the opposite direction!

There are a few reasons this may happen:

- Rotations don't map linearly to orientation, so interpolating them does not always result in the shortest path (i.e., to go from `270` to `0` degrees is not the same as going from `270` to `360`, even though the angles are equivalent).
- Gimbal lock is at play (first and last rotated axis align, so a degree of freedom is lost). See Wikipedia's page on Gimbal Lock [https://en.wikipedia.org/wiki/Gimbal_lock] for a detailed explanation of this problem.

Say no to Euler angles

The result of all this is that you should **not use** the `rotation` property of Node3D nodes in Godot for games. It's there to be used mainly in the editor, for coherence with the 2D engine, and for simple rotations (generally just one axis, or even two in limited cases). As much as you may be tempted, don't use it.

Instead, there is a better way to solve your rotation problems.

Introducing transforms

Godot uses the Transform3D datatype for orientations. Each Node3D node contains a `transform` property which is relative to the parent's transform, if the parent is a Node3D-derived type.

It is also possible to access the world coordinate transform via the `global_transform` property.

A transform has a Basis (transform.basis sub-property), which consists of three Vector3 vectors. These are accessed via the `transform.basis` property and can be accessed directly by `transform.basis.x`, `transform.basis.y`, and `transform.basis.z`. Each vector points in the direction its axis has been rotated, so they effectively describe the node's total rotation. The scale (as long as it's uniform) can also be inferred from the length of the axes. A *basis* can also be interpreted as a 3x3 matrix and used as `transform.basis[x][y]`.

A default basis (unmodified) is akin to:

GDScript

```
var basis = Basis()
# Contains the following default values:
basis.x = Vector3(1, 0, 0) # Vector pointing along the X axis
basis.y = Vector3(0, 1, 0) # Vector pointing along the Y axis
basis.z = Vector3(0, 0, 1) # Vector pointing along the Z axis
```

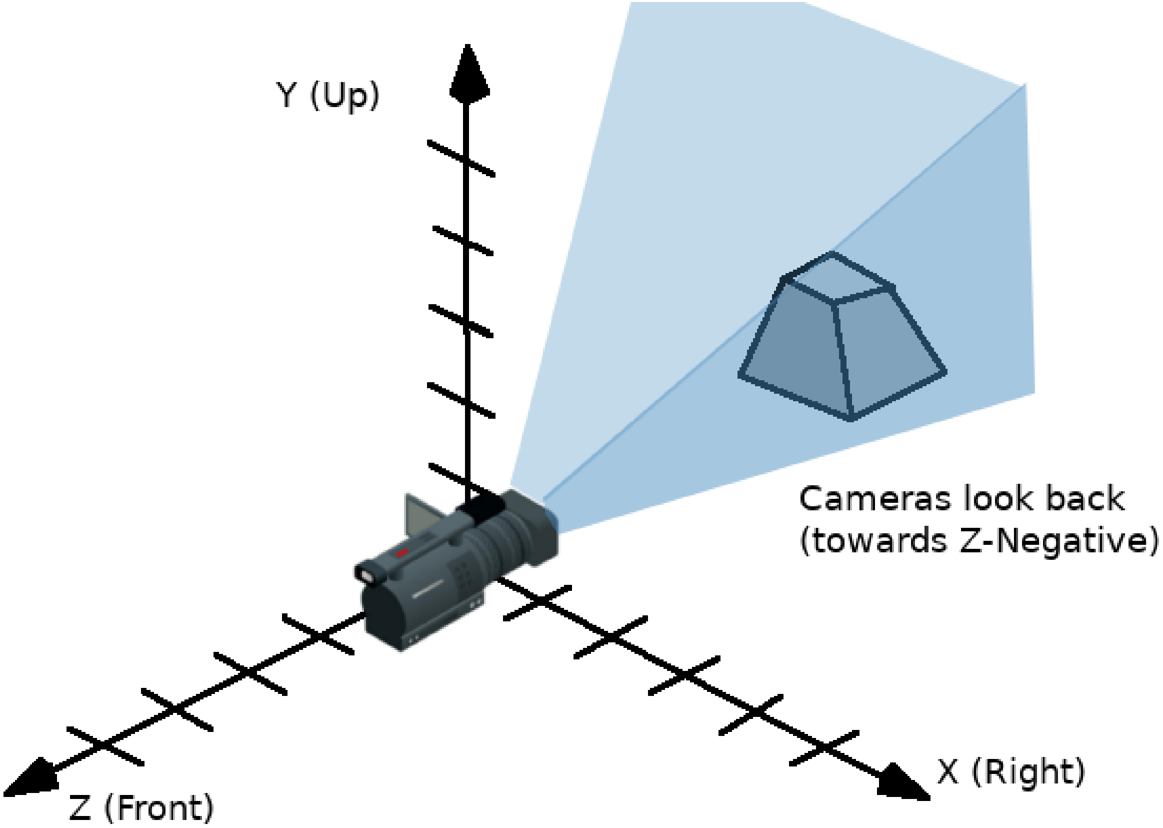
C#

```
// Due to technical limitations on structs in C# the default
// constructor will contain zero values for all fields.
var defaultBasis = new Basis();
GD.Print(defaultBasis); // prints: ((0, 0, 0), (0, 0, 0), (0,
0, 0))

// Instead we can use the Identity property.
var identityBasis = Basis.Identity;
GD.Print(identityBasis.X); // prints: (1, 0, 0)
GD.Print(identityBasis.Y); // prints: (0, 1, 0)
GD.Print(identityBasis.Z); // prints: (0, 0, 1)

// The Identity basis is equivalent to:
var basis = new Basis(Vector3.Right, Vector3.Up, Vector3.Back);
GD.Print(basis); // prints: ((1, 0, 0), (0, 1, 0), (0, 0, 1))
```

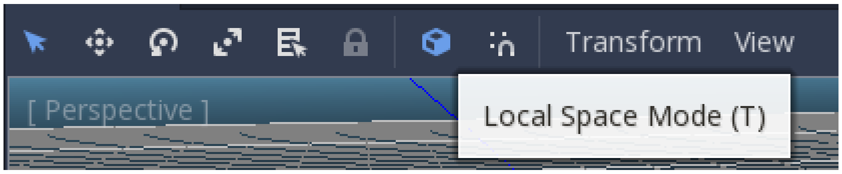
This is also an analog of a 3x3 identity matrix.

Following the OpenGL convention, `X` is the *Right* axis, `Y` is the *Up* axis and `Z` is the *Forward* axis.

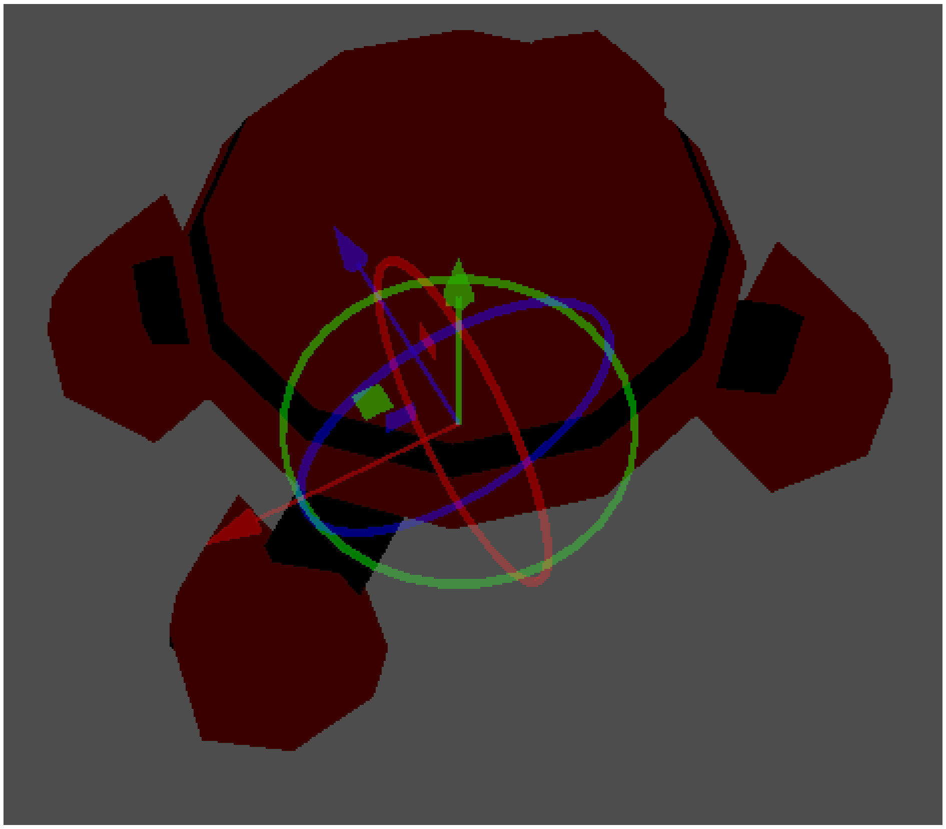
Together with the *basis*, a transform also has an *origin*. This is a *Vector3* specifying how far away from the actual origin `(0, 0, 0)` this transform is. Combining the *basis* with the *origin*, a *transform* efficiently represents a unique translation, rotation, and scale in space.


Y (Up)
Cameras look back (towards Z-Negative)
Z (Front)
X (Right)

One way to visualize a transform is to look at an object's 3D gizmo while in "local space" mode.


Transform
View
[Perspective]
Local Space Mode (T)

The gizmo's arrows show the `X`, `Y`, and `Z` axes (in red, green, and blue respectively) of the basis, while the gizmo's center is at the object's origin.



For more information on the mathematics of vectors and transforms, please read the Vector math tutorials.

Manipulating transforms

Of course, transforms are not as straightforward to manipulate as angles and have problems of their own.

It is possible to rotate a transform, either by multiplying its basis by another (this is called accumulation), or by using the rotation methods.

GDScript

```
var axis = Vector3(1, 0, 0) # Or Vector3.RIGHT
var rotation_amount = 0.1
# Rotate the transform around the X axis by 0.1 radians.
transform.basis = Basis(axis, rotation_amount) *
transform.basis
# shortened
transform.basis = transform.basis.rotated(axis,
rotation_amount)
```

C#

```
Transform3D transform = Transform;
Vector3 axis = new Vector3(1, 0, 0); // Or Vector3.Right
float rotationAmount = 0.1f;

// Rotate the transform around the X axis by 0.1 radians.
transform.Basis = new Basis(axis, rotationAmount) *
transform.Basis;
// shortened
transform.Basis = transform.Basis.Rotated(axis,
rotationAmount);

Transform = transform;
```

A method in Node3D simplifies this:

GDScript

```
# Rotate the transform around the X axis by 0.1 radians.
rotate(Vector3(1, 0, 0), 0.1)
# shortened
rotate_x(0.1)
```

C#

```
// Rotate the transform around the X axis by 0.1 radians.
Rotate(new Vector3(1, 0, 0), 0.1f);
// shortened
RotateX(0.1f);
```

This rotates the node relative to the parent node.

To rotate relative to object space (the node's own transform), use the following:

GDScript

```
# Rotate around the object's local X axis by 0.1 radians.
rotate_object_local(Vector3(1, 0, 0), 0.1)
```

C#

```
// Rotate around the object's local X axis by 0.1 radians.
RotateObjectLocal(new Vector3(1, 0, 0), 0.1f);
```

Precision errors

Doing successive operations on transforms will result in a loss of precision due to floating-point error. This means the scale of each axis may no longer be exactly `1.0`, and they may not be exactly `90` degrees from each other.

If a transform is rotated every frame, it will eventually start deforming over time. This is unavoidable.

There are two different ways to handle this. The first is to *orthonormalize* the transform after some time (maybe once per frame if you modify it every frame):

GDScript

```
transform = transform.orthonormalized()
```

C#

```
transform = transform.Orthonormalized();
```

This will make all axes have `1.0` length again and be `90` degrees from each other. However, any scale applied to the transform will be lost.

It is recommended you not scale nodes that are going to be manipulated; scale their children nodes instead (such as

MeshInstance3D). If you absolutely must scale the node, then re-apply it at the end:

GDScript

```
transform = transform.orthonormalized()
transform = transform.scaled(scale)
```

C#

```
transform = transform.Orthonormalized();
transform = transform.Scaled(scale);
```

Obtaining information

You might be thinking at this point: **"Ok, but how do I get angles from a transform?"**. The answer again is: you don't. You must do your best to stop thinking in angles.

Imagine you need to shoot a bullet in the direction your player is facing. Just use the forward axis (commonly `Z` or `-Z`).

GDScript

```
bullet.transform = transform
bullet.speed = transform.basis.z * BULLET_SPEED
```

C#

```
bullet.Transform = transform;
bullet.LinearVelocity = transform.Basis.Z * BulletSpeed;
```

Is the enemy looking at the player? Use the dot product for this (see the Vector math tutorial for an explanation of the dot product):

GDScript

```
# Get the direction vector from player to enemy
var direction = enemy.transform.origin -
```

```
player.transform.origin
if direction.dot(enemy.transform.basis.z) > 0:
    enemy.im_watching_you(player)
```

C#

```
// Get the direction vector from player to enemy
Vector3 direction = enemy.Transform.Origin -
player.Transform.Origin;
if (direction.Dot(enemy.Transform.Basis.Z) > 0)
{
    enemy.ImWatchingYou(player);
}
```

Strafe left:

GDScript

```
# Remember that +X is right
if Input.is_action_pressed("strafe_left"):
    translate_object_local(-transform.basis.x)
```

C#

```
// Remember that +X is right
if (Input.IsActionPressed("strafe_left"))
{
    TranslateObjectLocal(-Transform.Basis.X);
}
```

Jump:

GDScript

```
# Keep in mind Y is up-axis
if Input.is_action_just_pressed("jump"):
    velocity.y = JUMP_SPEED

move_and_slide()
```

C#

```
// Keep in mind Y is up-axis
if (Input.IsActionJustPressed("jump"))
    velocity.Y = JumpSpeed;

MoveAndSlide();
```

All common behaviors and logic can be done with just vectors.

Setting information

There are, of course, cases where you want to set information to a transform. Imagine a first person controller or orbiting camera. Those are definitely done using angles, because you *do want* the transforms to happen in a specific order.

For such cases, keep the angles and rotations *outside* the transform and set them every frame. Don't try to retrieve and reuse them because the transform is not meant to be used this way.

Example of looking around, FPS style:

GDScript

```
# accumulators
var rot_x = 0
var rot_y = 0

func _input(event):
    if event is InputEventMouseMotion and event.button_mask &
1:
        # modify accumulated mouse rotation
        rot_x += event.relative.x * LOOKAROUND_SPEED
        rot_y += event.relative.y * LOOKAROUND_SPEED
        transform.basis = Basis() # reset rotation
        rotate_object_local(Vector3(0, 1, 0), rot_x) # first
rotate in Y
```

```
        rotate_object_local(Vector3(1, 0, 0), rot_y) # then rotate in X
```

C#

```
// accumulators
private float _rotationX = 0f;
private float _rotationY = 0f;

public override void _Input(InputEvent @event)
{
    if (@event is InputEventMouseMotion mouseMotion)
    {
        // modify accumulated mouse rotation
        _rotationX += mouseMotion.Relative.X * LookAroundSpeed;
        _rotationY += mouseMotion.Relative.Y * LookAroundSpeed;

        // reset rotation
        Transform3D transform = Transform;
        transform.Basis = Basis.Identity;
        Transform = transform;

        RotateObjectLocal(Vector3.Up, _rotationX); // first rotate about Y
        RotateObjectLocal(Vector3.Right, _rotationY); // then rotate about X
    }
}
```

As you can see, in such cases it's even simpler to keep the rotation outside, then use the transform as the *final* orientation.

Interpolating with quaternions

Interpolating between two transforms can efficiently be done with quaternions. More information about how quaternions work can be found in other places around the Internet. For practical use, it's enough to understand that pretty much their main use is doing a closest path interpolation. As in, if you have two rotations, a quaternion will smoothly allow interpolation between them using the closest axis.

Converting a rotation to quaternion is straightforward.

GDScript

```
# Convert basis to quaternion, keep in mind scale is lost
var a = Quaternion(transform.basis)
var b = Quaternion(transform2.basis)
# Interpolate using spherical-linear interpolation (SLERP).
var c = a.slerp(b,0.5) # find halfway point between a and b
# Apply back
transform.basis = Basis(c)
```

C#

```
// Convert basis to quaternion, keep in mind scale is lost
var a = transform.Basis.GetQuaternion();
var b = transform2.Basis.GetQuaternion();
// Interpolate using spherical-linear interpolation (SLERP).
var c = a.Slerp(b, 0.5f); // find halfway point between a and b
// Apply back
transform.Basis = new Basis(c);
```

The Quaternion type reference has more information on the datatype (it can also do transform accumulation, transform points, etc., though this is used less often). If you interpolate or apply operations to quaternions many times, keep in mind they need to be eventually normalized. Otherwise, they will also suffer from numerical precision errors.

Quaternions are useful when doing camera/path/etc. interpolations, as the result will always be correct and smooth.

Transforms are your friend

For most beginners, getting used to working with transforms can take some time. However, once you get used to them, you will appreciate their simplicity and power.

Don't hesitate to ask for help on this topic in any of Godot's online communities [https://godotengine.org/community] and, once you become confident enough, please help others!

Procedural geometry

There are many ways to procedurally generate geometry in Godot. In this tutorial series, we will explore a few of them. Each technique has its own benefits and drawbacks, so it is best to understand each one and how it can be useful in a given situation.

- Using the ArrayMesh
- Using the MeshDataTool
- Using the SurfaceTool
- Using ImmediateMesh

Note

All the procedural geometry generation methods described here run on the CPU. Godot doesn't support generating geometry on the GPU yet.

What is geometry?

Geometry is a fancy way of saying shape. In computer graphics, geometry is typically represented by an array of positions called "vertices". In Godot, geometry is represented by Meshes.

What is a Mesh?

Many things in Godot have mesh in their name: the Mesh, the ArrayMesh, the ImmediateMesh, the MeshInstance3D, the MultiMesh, and the MultiMeshInstance3D. While they are all related, they have slightly different uses.

Meshes and ArrayMeshes are resources that are drawn using a MeshInstance3D node. Resources like Meshes and ArrayMeshes cannot be added to the scene directly. A MeshInstance3D represents one instance of a mesh in your scene. You can reuse a single mesh in multiple MeshInstance3Ds to draw it in different parts of your scene with different materials or transformations (scale, rotation, position etc.).

If you are going to draw the same object many times, it can be helpful to use a MultiMesh with a MultiMeshInstance3D. MultiMeshInstance3Ds draw meshes thousands of times very cheaply by taking advantage of hardware instancing. The drawback with using a MultiMeshInstance3D is that each of your mesh's surfaces are limited to one material for all instances. It uses an instance array to store different colors and transformations for each instance, but all the instances of each surface use the same material.

What a Mesh is

A Mesh is composed of one or more surfaces. A surface is an array composed of multiple sub-arrays containing vertices, normals, UVs, etc. Normally the process of constructing surfaces and meshes is hidden from the user in the RenderingServer, but with ArrayMeshes, the user can construct a Mesh manually by passing in an array containing the surface information.

Surfaces

Each surface has its own material. Alternatively, you can override the material for all surfaces in the Mesh when you use a MeshInstance3D using the material_override property.

Surface array

The surface array is an array of length `ArrayMesh.ARRAY_MAX`. Each position in the array is filled with a sub-array containing per-vertex information. For example, the array located at `ArrayMesh.ARRAY_NORMAL` is a PackedVector3Array of vertex normals. See Mesh.ArrayType for more information.

The surface array can be indexed or non-indexed. Creating a non-indexed array is as easy as not assigning an array at the index `ArrayMesh.ARRAY_INDEX`. A non-indexed array stores unique vertex information for every triangle, meaning that when two triangles share a vertex, the vertex is duplicated in the array. An indexed surface array only stores vertex information for each unique vertex and then also stores an array of indices which maps out how to construct the triangles from the vertex array. In general, using an indexed array is faster, but it means you have to share vertex data between triangles, which is not always desired (e.g. when you want per-face normals).

Tools

Godot provides different ways of accessing and working with geometry. More information on each will be provided in the following tutorials.

ArrayMesh

The ArrayMesh resource extends Mesh to add a few different quality of life functions and, most importantly, the ability to construct a Mesh surface through scripting.

For more information about the ArrayMesh, please see the ArrayMesh tutorial.

MeshDataTool

The MeshDataTool is a resource that converts Mesh data into arrays of vertices, faces, and edges that can be modified at runtime.

For more information about the MeshDataTool, please see the MeshDataTool tutorial.

SurfaceTool

The SurfaceTool allows the creation of Meshes using an OpenGL 1.x immediate mode style interface.

For more information about the SurfaceTool, please see the SurfaceTool tutorial.

ImmediateMesh

ImmediateMesh is a mesh that uses an immediate mode style interface (like SurfaceTool) to draw objects. The difference between ImmediateMesh and the SurfaceTool is that ImmediateMesh is drawn directly with code dynamically, while the SurfaceTool is used to generate a Mesh that you can do whatever you want with.

ImmediateMesh is useful for prototyping because of its straightforward API, but it is slow because the geometry is rebuilt each time you make a change. It is most useful for adding simple geometry for visual debugging (e.g. by drawing lines to visualize physics raycasts etc.).

For more information about ImmediateMesh, please see the ImmediateMesh tutorial.

Which one should I use?

Which approach you use depends on what you are trying to do and what kind of procedure you are comfortable with.

Both SurfaceTool and ArrayMesh are excellent for generating static geometry (meshes) that don't change over time.

Using an ArrayMesh is slightly faster than using a SurfaceTool, but the API is a little more challenging. Additionally, SurfaceTool has a few quality of life methods such as `generate_normals()` and `index()`.

ImmediateMesh is more limited than both ArrayMesh and SurfaceTool. However, if you need the geometry to change every frame anyway, it provides a much easier interface that can be slightly faster than generating an ArrayMesh every frame.

The MeshDataTool is not fast, but it gives you access to all kinds of properties of the mesh that you don't get with the others (edges, faces, etc.). It is incredibly useful when you need that sort of data to transform the mesh, but it is not a good idea to use it if that extra information is not needed. The MeshDataTool is best used if you are going to be using an algorithm that requires access to the face or edge array.

Using the ArrayMesh

This tutorial will present the basics of using an ArrayMesh.

To do so, we will use the function add_surface_from_arrays(), which takes up to five parameters. The first two are required, while the last three are optional.

The first parameter is the `PrimitiveType`, an OpenGL concept that instructs the GPU how to arrange the primitive based on the vertices given, i.e. whether they represent triangles, lines, points, etc. See Mesh.PrimitiveType for the options available.

The second parameter, `arrays`, is the actual Array that stores the mesh information. The array is a normal Godot array that is constructed with empty brackets `[]`. It stores a `Packed**Array` (e.g. PackedVector3Array, PackedInt32Array, etc.) for each type of information that will be used to build the surface.

Common elements of `arrays` are listed below, together with the position they must have within `arrays`. See Mesh.ArrayType for a full list.

Index	Mesh.ArrayType Enum	Array type
0	`ARRAY_VERTEX`	PackedVector3Array or PackedVector2Array
1	`ARRAY_NORMAL`	PackedVector3Array

Index	Mesh.ArrayType Enum	Array type
2	`ARRAY_TANGENT`	PackedFloat32Array or PackedFloat64Array of groups of 4 floats. The first 3 floats determine the tangent, and the last float the binormal direction as -1 or 1.
3	`ARRAY_COLOR`	PackedColorArray
4	`ARRAY_TEX_UV`	PackedVector2Array or PackedVector3Array
5	`ARRAY_TEX_UV2`	PackedVector2Array or PackedVector3Array
10	`ARRAY_BONES`	PackedFloat32Array of groups of 4 floats or PackedInt32Array of groups of 4 ints. Each group lists indexes of 4 bones that affects a given vertex.
11	`ARRAY_WEIGHTS`	PackedFloat32Array or PackedFloat64Array of groups of 4 floats. Each float lists the amount of weight the corresponding bone in `ARRAY_BONES` has on a given vertex.
12	`ARRAY_INDEX`	PackedInt32Array

Index	Mesh.ArrayType Enum	Array type

In most cases when creating a mesh, we define it by its vertex positions. So usually, the array of vertices (at index 0) is required, while the index array (at index 12) is optional and will only be used if included. It is also possible to create a mesh with only the index array and no vertex array, but that's beyond the scope of this tutorial.

All the other arrays carry information about the vertices. They are optional and will only be used if included. Some of these arrays (e.g. `ARRAY_COLOR`) use one entry per vertex to provide extra information about vertices. They must have the same size as the vertex array. Other arrays (e.g. `ARRAY_TANGENT`) use four entries to describe a single vertex. These must be exactly four times larger than the vertex array.

For normal usage, the last three parameters in add_surface_from_arrays() are typically left empty.

Setting up the ArrayMesh

In the editor, create a MeshInstance3D and add an ArrayMesh to it in the Inspector. Normally, adding an ArrayMesh in the editor is not useful, but in this case it allows us to access the ArrayMesh from code without creating one.

Next, add a script to the MeshInstance3D.

Under `_ready()`, create a new Array.

GDScript

```
var surface_array = []
```

C#

```
var surfaceArray = new Godot.Collections.Array();
```

This will be the array that we keep our surface information in - it will hold all the arrays of data that the surface needs. Godot will expect it to be of size `Mesh.ARRAY_MAX`, so resize it accordingly.

GDScript

```
var surface_array = []
surface_array.resize(Mesh.ARRAY_MAX)
```

C#

```
var surfaceArray = new Godot.Collections.Array();
surfaceArray.Resize((int)Mesh.ArrayType.Max);
```

Next create the arrays for each data type you will use.

GDScript

```
var verts = PackedVector3Array()
var uvs = PackedVector2Array()
var normals = PackedVector3Array()
var indices = PackedInt32Array()
```

C#

```
var verts = new List<Vector3>();
var uvs = new List<Vector2>();
var normals = new List<Vector3>();
var indices = new List<int>();
```

Once you have filled your data arrays with your geometry you can create a mesh by adding each array to `surface_array` and then committing to the mesh.

GDScript

```
surface_array[Mesh.ARRAY_VERTEX] = verts
surface_array[Mesh.ARRAY_TEX_UV] = uvs
surface_array[Mesh.ARRAY_NORMAL] = normals
surface_array[Mesh.ARRAY_INDEX] = indices

# No blendshapes, lods, or compression used.
mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
surface_array)
```

C#

```
surfaceArray[(int)Mesh.ArrayType.Vertex] = verts.ToArray();
surfaceArray[(int)Mesh.ArrayType.TexUV] = uvs.ToArray();
surfaceArray[(int)Mesh.ArrayType.Normal] = normals.ToArray();
surfaceArray[(int)Mesh.ArrayType.Index] = indices.ToArray();

var arrMesh = Mesh as ArrayMesh;
if (arrMesh != null)
{
    // No blendshapes, lods, or compression used.
    arrMesh.AddSurfaceFromArrays(Mesh.PrimitiveType.Triangles,
surfaceArray);
}
```

Note

In this example, we used `Mesh.PRIMITIVE_TRIANGLES`, but you can use any primitive type available from mesh.

Put together, the full code looks like:

GDScript

```
extends MeshInstance3D

func _ready():
    var surface_array = []
    surface_array.resize(Mesh.ARRAY_MAX)
    # PackedVector**Arrays for mesh construction.
    var verts = PackedVector3Array()
    var uvs = PackedVector2Array()
    var normals = PackedVector3Array()
    var indices = PackedInt32Array()

    #######################################
    ## Insert code here to generate mesh ##
    #######################################

    # Assign arrays to surface array.
    surface_array[Mesh.ARRAY_VERTEX] = verts
    surface_array[Mesh.ARRAY_TEX_UV] = uvs
    surface_array[Mesh.ARRAY_NORMAL] = normals
    surface_array[Mesh.ARRAY_INDEX] = indices

    # Create mesh surface from mesh array.
    # No blendshapes, lods, or compression used.
    mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
surface_array)
```

C#

```
public partial class MyMeshInstance3D : MeshInstance3D
{
    public override void _Ready()
    {
        var surfaceArray = new Godot.Collections.Array();
        surfaceArray.Resize((int)Mesh.ArrayType.Max);

        // C# arrays cannot be resized or expanded, so use
Lists to create geometry.
        var verts = new List<Vector3>();
        var uvs = new List<Vector2>();
        var normals = new List<Vector3>();
        var indices = new List<int>();

        /***********************************
        * Insert code here to generate mesh.
        * *********************************/
        // Convert Lists to arrays and assign to surface array
        surfaceArray[(int)Mesh.ArrayType.Vertex] =
verts.ToArray();
        surfaceArray[(int)Mesh.ArrayType.TexUV] =
uvs.ToArray();
        surfaceArray[(int)Mesh.ArrayType.Normal] =
normals.ToArray();
        surfaceArray[(int)Mesh.ArrayType.Index] =
indices.ToArray();

        var arrMesh = Mesh as ArrayMesh;
        if (arrMesh != null)
        {
            // Create mesh surface from mesh array
            // No blendshapes, lods, or compression used.

arrMesh.AddSurfaceFromArrays(Mesh.PrimitiveType.Triangles,
surfaceArray);
        }
    }
}
```

The code that goes in the middle can be whatever you want. Below we will present some example code for generating a sphere.

Generating geometry

Here is sample code for generating a sphere. Although the code is presented in GDScript, there is nothing Godot specific about the approach to generating it. This implementation has nothing in particular to do with ArrayMeshes and is just a generic approach to generating a sphere. If you are having trouble understanding it or want to learn more about procedural geometry in general, you can use any tutorial that you find online.

GDScript

```
extends MeshInstance3D

var rings = 50
var radial_segments = 50
var radius = 1

func _ready():

    # Insert setting up the PackedVector**Arrays here.

    # Vertex indices.
    var thisrow = 0
    var prevrow = 0
    var point = 0

    # Loop over rings.
    for i in range(rings + 1):
        var v = float(i) / rings
        var w = sin(PI * v)
        var y = cos(PI * v)

        # Loop over segments in ring.
        for j in range(radial_segments + 1):
            var u = float(j) / radial_segments
            var x = sin(u * PI * 2.0)
            var z = cos(u * PI * 2.0)
            var vert = Vector3(x * radius * w, y * radius, z *
radius * w)
            verts.append(vert)
            normals.append(vert.normalized())
            uvs.append(Vector2(u, v))
            point += 1

            # Create triangles in ring using indices.
            if i > 0 and j > 0:
                indices.append(prevrow + j - 1)
                indices.append(prevrow + j)
                indices.append(thisrow + j - 1)

                indices.append(prevrow + j)
                indices.append(thisrow + j)
                indices.append(thisrow + j - 1)
        prevrow = thisrow
        thisrow = point

  # Insert committing to the ArrayMesh here.
```

C#

```
public partial class MyMeshInstance3D : MeshInstance3D
{
    private int _rings = 50;
    private int _radialSegments = 50;
    private float _radius = 1;

    public override void _Ready()
    {
        // Insert setting up the surface array and lists here.

        // Vertex indices.
        var thisRow = 0;
        var prevRow = 0;
        var point = 0;

        // Loop over rings.
        for (var i = 0; i < _rings + 1; i++)
        {
            var v = ((float)i) / _rings;
            var w = Mathf.Sin(Mathf.Pi * v);
            var y = Mathf.Cos(Mathf.Pi * v);

            // Loop over segments in ring.
            for (var j = 0; j < _radialSegments + 1; j++)
            {
                var u = ((float)j) / _radialSegments;
                var x = Mathf.Sin(u * Mathf.Pi * 2);
                var z = Mathf.Cos(u * Mathf.Pi * 2);
                var vert = new Vector3(x * _radius * w, y *
_radius, z * _radius * w);
                verts.Add(vert);
                normals.Add(vert.Normalized());
                uvs.Add(new Vector2(u, v));
                point += 1;
                // Create triangles in ring using indices.
                if (i > 0 && j > 0)
                {
                    indices.Add(prevRow + j - 1);
                    indices.Add(prevRow + j);
                    indices.Add(thisRow + j - 1);

                    indices.Add(prevRow + j);
                    indices.Add(thisRow + j);
                    indices.Add(thisRow + j - 1);
                }
            }

            prevRow = thisRow;
            thisRow = point;
        }

        // Insert committing to the ArrayMesh here.
    }
}
```

Saving

Finally, we can use the ResourceSaver class to save the ArrayMesh. This is useful when you want to generate a mesh and then use it later without having to re-generate it.

GDScript

```
# Saves mesh to a .tres file with compression enabled.
ResourceSaver.save(mesh, "res://sphere.tres",
ResourceSaver.FLAG_COMPRESS)
```

C#

```
// Saves mesh to a .tres file with compression enabled.
ResourceSaver.Save(Mesh, "res://sphere.tres",
ResourceSaver.SaverFlags.Compress);
```

Using the MeshDataTool

The MeshDataTool is not used to generate geometry. But it is helpful for dynamically altering geometry, for example if you want to write a script to tessellate, simplify, or deform meshes.

The MeshDataTool is not as fast as altering arrays directly using ArrayMesh. However, it provides more information and tools to work with meshes than the ArrayMesh does. When the MeshDataTool is used, it calculates mesh data that is not available in ArrayMeshes such as faces and edges, which are necessary for certain mesh algorithms. If you do not need this extra information then it may be better to use an ArrayMesh.

Note

MeshDataTool can only be used on Meshes that use the PrimitiveType `Mesh.PRIMITIVE_TRIANGLES`.

We initialize the MeshDataTool from an ArrayMesh by calling `create_from_surface()`. If there is already data initialized in the MeshDataTool, calling `create_from_surface()` will clear it for you. Alternatively, you can call `clear()` yourself before re-using the MeshDataTool.

In the examples below, assume an ArrayMesh called `mesh` has already been created. See ArrayMesh tutorial for an example of mesh generation.

GDScript

```
var mdt = MeshDataTool.new()
```

```
mdt.create_from_surface(mesh, 0)
```

`create_from_surface()` uses the vertex arrays from the ArrayMesh to calculate two additional arrays, one for edges and one for faces, for a total of three arrays.

An edge is a connection between any two vertices. Each edge in the edge array contains a reference to the two vertices it is composed of, and up to two faces that it is contained within.

A face is a triangle made up of three vertices and three corresponding edges. Each face in the face array contains a reference to the three vertices and three edges it is composed of.

The vertex array contains edge, face, normal, color, tangent, uv, uv2, bone, and weight information connected with each vertex.

To access information from these arrays you use a function of the form `get_****()`:

GDScript

```
mdt.get_vertex_count() # Returns number of vertices in vertex array.
mdt.get_vertex_faces(0) # Returns array of faces that contain vertex[0].
mdt.get_face_normal(1) # Calculates and returns face normal of the second face.
mdt.get_edge_vertex(10, 1) # Returns the second vertex comprising the edge at index 10.
```

What you choose to do with these functions is up to you. A common use case is to iterate over all vertices and transform them in some way:

GDScript

```
for i in range(get_vertex_count):
    var vert = mdt.get_vertex(i)
    vert *= 2.0 # Scales the vertex by doubling size.
    mdt.set_vertex(i, vert)
```

These modifications are not done in place on the ArrayMesh. If you are dynamically updating an existing ArrayMesh, first delete the existing surface before adding a new one using commit_to_surface():

GDScript

```
mesh.clear_surfaces() # Deletes all of the mesh's surfaces.
mdt.commit_to_surface(mesh)
```

Below is a complete example that turns a spherical mesh called `mesh` into a randomly deformed blob complete with updated normals and vertex colors. See ArrayMesh tutorial for how to generate the base mesh.

GDScript

```
extends MeshInstance3D

var fnl = FastNoiseLite.new()
var mdt = MeshDataTool.new()

func _ready():
    fnl.frequency = 0.7

    mdt.create_from_surface(mesh, 0)

    for i in range(mdt.get_vertex_count()):
        var vertex = mdt.get_vertex(i).normalized()
        # Push out vertex by noise.
        vertex = vertex * (fnl.get_noise_3dv(vertex) * 0.5 +
0.75)
        mdt.set_vertex(i, vertex)

    # Calculate vertex normals, face-by-face.
    for i in range(mdt.get_face_count()):
        # Get the index in the vertex array.
        var a = mdt.get_face_vertex(i, 0)
        var b = mdt.get_face_vertex(i, 1)
        var c = mdt.get_face_vertex(i, 2)
        # Get vertex position using vertex index.
        var ap = mdt.get_vertex(a)
        var bp = mdt.get_vertex(b)
        var cp = mdt.get_vertex(c)
        # Calculate face normal.
        var n = (bp - cp).cross(ap - bp).normalized()
        # Add face normal to current vertex normal.
        # This will not result in perfect normals, but it will
be close.
        mdt.set_vertex_normal(a, n + mdt.get_vertex_normal(a))
        mdt.set_vertex_normal(b, n + mdt.get_vertex_normal(b))
        mdt.set_vertex_normal(c, n + mdt.get_vertex_normal(c))

    # Run through vertices one last time to normalize normals
and
    # set color to normal.
    for i in range(mdt.get_vertex_count()):
        var v = mdt.get_vertex_normal(i).normalized()
        mdt.set_vertex_normal(i, v)
        mdt.set_vertex_color(i, Color(v.x, v.y, v.z))

    mesh.clear_surfaces()
    mdt.commit_to_surface(mesh)
```

Using the SurfaceTool

The SurfaceTool provides a useful interface for constructing geometry. The interface is similar to the ImmediateMesh class. You set each per-vertex attribute (e.g. normal, uv, color) and then when you add a vertex it captures the attributes.

The SurfaceTool also provides some useful helper functions like `index()` and `generate_normals()`.

Attributes are added before each vertex is added:

GDScript

```
st.set_normal() # Overwritten by normal below.
st.set_normal() # Added to next vertex.
st.set_color() # Added to next vertex.
st.add_vertex() # Captures normal and color above.
st.set_normal() # Normal never added to a vertex.
```

C#

```
st.SetNormal(); // Overwritten by normal below.
st.SetNormal(); // Added to next vertex.
st.SetColor(); // Added to next vertex.
st.AddVertex(); // Captures normal and color above.
st.SetNormal(); // Normal never added to a vertex.
```

When finished generating your geometry with the SurfaceTool call `commit()` to finish generating the mesh. If an ArrayMesh is passed to `commit()` then it appends a new surface to the end of the ArrayMesh. While if nothing is passed in, `commit()` returns an ArrayMesh.

GDScript

```
st.commit(mesh)
# Or:
```

```
var mesh = st.commit()
```

C#

```
st.Commit(mesh);
// Or:
var mesh = st.Commit();
```

Code creates a triangle with indices

GDScript

```
var st = SurfaceTool.new()

st.begin(Mesh.PRIMITIVE_TRIANGLES)

# Prepare attributes for add_vertex.
st.set_normal(Vector3(0, 0, 1))
st.set_uv(Vector2(0, 0))
# Call last for each vertex, adds the above attributes.
st.add_vertex(Vector3(-1, -1, 0))

st.set_normal(Vector3(0, 0, 1))
st.set_uv(Vector2(0, 1))
st.add_vertex(Vector3(-1, 1, 0))

st.set_normal(Vector3(0, 0, 1))
st.set_uv(Vector2(1, 1))
st.add_vertex(Vector3(1, 1, 0))

# Commit to a mesh.
var mesh = st.commit()
```

C#

```
var st = new SurfaceTool();

st.Begin(Mesh.PrimitiveType.Triangles);

// Prepare attributes for AddVertex.
st.SetNormal(new Vector3(0, 0, 1));
st.SetUV(new Vector2(0, 0));
// Call last for each vertex, adds the above attributes.
st.AddVertex(new Vector3(-1, -1, 0));

st.SetNormal(new Vector3(0, 0, 1));
st.SetUV(new Vector2(0, 1));
st.AddVertex(new Vector3(-1, 1, 0));

st.SetNormal(new Vector3(0, 0, 1));
st.SetUV(new Vector2(1, 1));
st.AddVertex(new Vector3(1, 1, 0));

// Commit to a mesh.
var mesh = st.Commit();
```

You can optionally add an index array, either by calling `add_index()` and adding vertices to the index array or by calling `index()` which shrinks the vertex array to remove duplicate vertices.

GDScript

```
# Creates a quad from four corner vertices.
# add_index does not need to be called before add_vertex.
st.add_index(0)
st.add_index(1)
st.add_index(2)

st.add_index(1)
st.add_index(3)
st.add_index(2)

# Alternatively:
st.index()
```

C#

```
// Creates a quad from four corner vertices.
// AddIndex does not need to be called before AddVertex.
st.AddIndex(0);
st.AddIndex(1);
st.AddIndex(2);
st.AddIndex(1);
st.AddIndex(3);
st.AddIndex(2);

// Alternatively:
st.Index();
```

Similarly, if you have an index array, but you want each vertex to be unique (e.g. because you want to use unique normals or colors per face instead of per-vertex), you can call `deindex()`.

GDScript

```
st.deindex()
```

C#

```
st.Deindex();
```

If you don't add custom normals yourself, you can add them using `generate_normals()`, which should be called after generating geometry and before committing the mesh using `commit()` or `commit_to_arrays()`. Calling `generate_normals(true)` will flip the resulting normals. As a side note, `generate_normals()` only works if the primitive type is set to `Mesh.PRIMITIVE_TRIANGLES`.

You may notice that normal mapping or other material properties look broken on the generated mesh. This is because normal mapping **requires** the mesh to feature *tangents*, which are separate from *normals*. You can either add custom tangents manually, or generate them automatically with `generate_tangents()`. This method requires that each vertex have UVs and normals set already.

GDScript

```
st.generate_normals()
st.generate_tangents()
```

C#

```
st.GenerateNormals();
st.GenerateTangents();
```

By default, when generating normals, they will be calculated on a per-face basis. If you want smooth vertex normals, when adding vertices, call `add_smooth_group()`. `add_smooth_group()` needs to be called while building the geometry, e.g. before the call to `add_vertex()` (if non-indexed) or `add_index()` (if indexed).

Using ImmediateMesh

The ImmediateMesh is a convenient tool to create dynamic geometry using an OpenGL 1.x-style API. Which makes it both approachable to use and efficient for meshes which need to be updated every frame.

Generating complex geometry (several thousand vertices) with this tool is inefficient, even if it's done only once. Instead, it is designed to generate simple geometry that changes every frame.

First, you need to create a MeshInstance3D and add an ImmediateMesh to it in the Inspector.

Next, add a script to the MeshInstance3D. The code for the ImmediateMesh should go in the `_process()` function if you want it to update each frame, or in the `_ready()` function if you want to create the mesh once and not update it. If you only generate a surface once, the ImmediateMesh is just as efficient as any other kind of mesh as the generated mesh is cached and reused.

To begin generating geometry you must call `surface_begin()`. `surface_begin()` takes a `PrimitiveType` as an argument. `PrimitiveType` instructs the GPU how to arrange the primitive based on the vertices given whether it is triangles, lines, points, etc. A complete list can be found under the Mesh class reference page.

Once you have called `surface_begin()` you are ready to start adding vertices. You add vertices one at a time. First you add vertex specific attributes such as normals or UVs using `surface_set_****()` (e.g. `surface_set_normal()`). Then you call `surface_add_vertex()` to add a vertex with those attributes. For example:

GDScript

```
# Add a vertex with normal and uv.
surface_set_normal(Vector3(0, 1, 0))
surface_set_uv(Vector2(1, 1))
surface_add_vertex(Vector3(0, 0, 1))
```

Only attributes added before the call to `surface_add_vertex()` will be included in that vertex. If you add an attribute twice before calling `surface_add_vertex()`, only the second call will be used.

Finally, once you have added all your vertices call `surface_end()` to signal that you have finished generating the surface. You can call `surface_begin()` and `surface_end()` multiple times to generate multiple surfaces for the mesh.

The example code below draws a single triangle in the `_ready()` function.

GDScript

```
extends MeshInstance3D

func _ready():
    # Begin draw.
    mesh.surface_begin(Mesh.PRIMITIVE_TRIANGLES)

    # Prepare attributes for add_vertex.
    mesh.surface_set_normal(Vector3(0, 0, 1))
    mesh.surface_set_uv(Vector2(0, 0))
    # Call last for each vertex, adds the above attributes.
    mesh.surface_add_vertex(Vector3(-1, -1, 0))

    mesh.surface_set_normal(Vector3(0, 0, 1))
    mesh.surface_set_uv(Vector2(0, 1))
    mesh.surface_add_vertex(Vector3(-1, 1, 0))

    mesh.surface_set_normal(Vector3(0, 0, 1))
    mesh.surface_set_uv(Vector2(1, 1))
    mesh.surface_add_vertex(Vector3(1, 1, 0))
    # End drawing.
    mesh.surface_end()
```

The ImmediateMesh can also be used across frames. Each time you call `surface_begin()` and `surface_end()`, you are adding a new surface to the ImmediateMesh. If you want to recreate the mesh from scratch each frame, call `clear_surfaces()` before calling `surface_begin()`.

GDScript

```
extends MeshInstance3D

func _process(delta):

    # Clean up before drawing.
    mesh.clear_surfaces()

    # Begin draw.
    mesh.surface_begin(Mesh.PRIMITIVE_TRIANGLES)

    # Draw mesh.

    # End drawing.
    mesh.surface_end()
```

The above code will dynamically create and draw a single surface each frame.

3D text

Introduction

In a project, there may be times when text needs to be created as part of a 3D scene and not just in the HUD. Godot provides 2 methods to do this: the Label3D node and the TextMesh *resource* for a MeshInstance3D node.

Additionally, Godot makes it possible to position Control nodes according to a 3D point's position on the camera. This can be used as an alternative to "true" 3D text in situations where Label3D and TextMesh aren't flexible enough.

See also

You can see 3D text in action using the 3D Labels and Texts demo project [https://github.com/godotengine/godot-demo-projects/tree/master/3d/labels_and_texts].

This page does **not** cover how to display a GUI scene within a 3D environment. For information on how to achieve that, see the GUI in 3D [https://github.com/godotengine/godot-demo-projects/tree/master/viewport/gui_in_3d] demo project.

Label3D

Label3D text

Label3D behaves like a Label node, but in 3D space. Unlike the Label node, this Label3D node does **not** inherit properties of a GUI theme. However, its look remains customizable and uses the same font subresource as Control nodes (including support for MSDF font rendering).

Advantages

- Label3D is faster to generate than TextMesh. While both use a caching mechanism to only render new glyphs once, Label3D will still be faster to (re)generate, especially for long text. This can avoid stuttering during gameplay on low-end CPUs or mobile.
- Label3D can use bitmap fonts and dynamic fonts (with and without MSDF or mipmaps). This makes it more flexible on that aspect compared to TextMesh, especially for rendering fonts with self-intersecting outlines or colored fonts (emoji).

See also

See Using Fonts for guidelines on configuring font imports.

Limitations

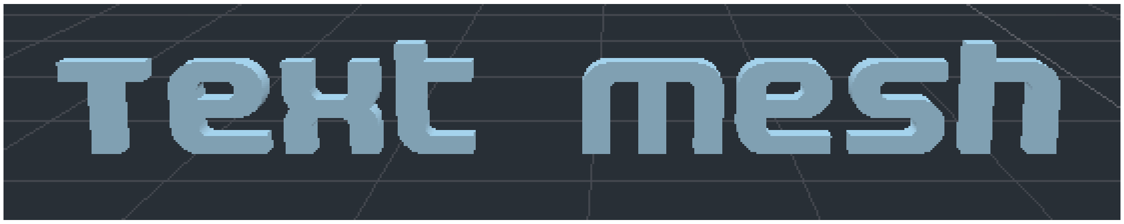
By default, Label3D has limited interaction with a 3D environment. It can be occluded by geometry and lit by light sources if the **Shaded** flag is enabled. However, it will not cast shadows even if **Cast Shadow** is set to **On** in the Label3D's GeometryInstance3D properties. This is because the node internally generates a quad mesh (one glyph per quad) with transparent textures and has the same limitations as Sprite3D. Transparency sorting issues can also become apparent when several Label3Ds overlap, especially if they have outlines.

This can be mitigated by setting the Label3D's transparency mode to **Alpha Cut**, at the cost of less smooth text rendering. The **Opaque Pre-Pass** transparency mode can preserve text smoothness while allowing the Label3D to cast shadows, but some transparency sorting issues will remain.

See Transparency sorting section in the 3D rendering limitations page for more information.

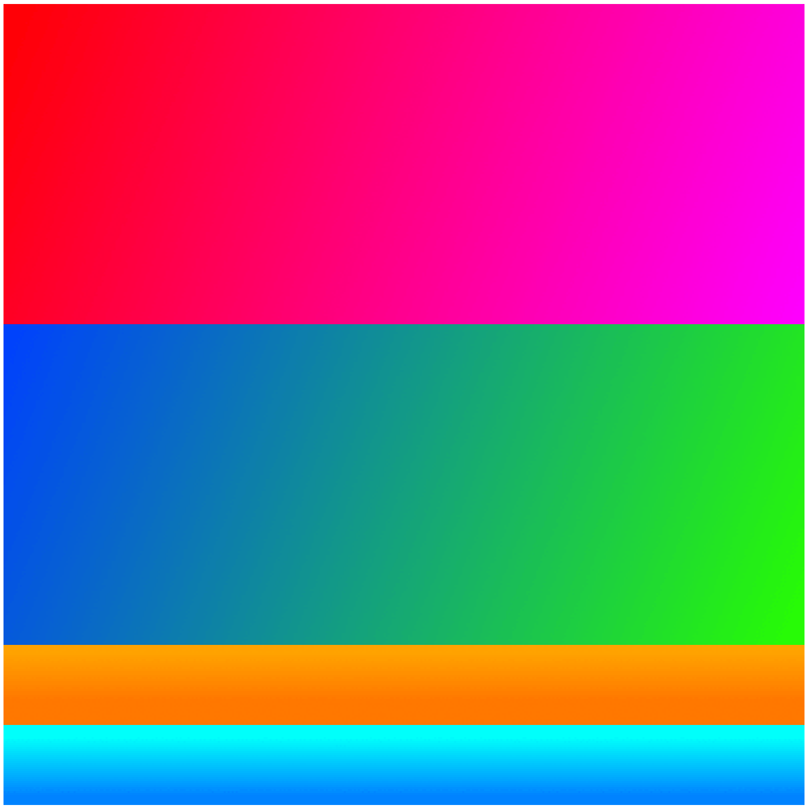
Text rendering quality can also suffer when the Label3D is viewed at a distance. To improve text rendering quality, enable mipmaps on the font or switch the font to use MSDF rendering.

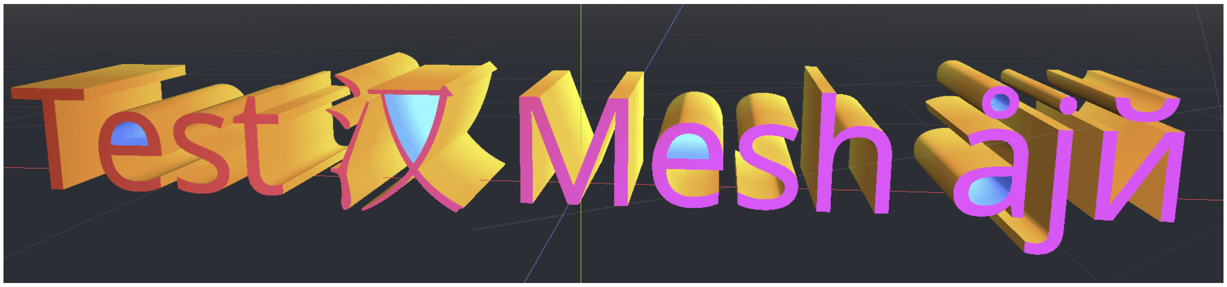
TextMesh



The TextMesh resource has similarities to Label3D. They both display text in a 3D scene, and will use the same font subresource. However, instead of generating transparent quads, TextMesh generates 3D geometry that represents the glyphs' contours and has the properties of a mesh. As a result, a TextMesh is shaded by default and automatically casts shadows onto the environment. A TextMesh can also have a material applied to it (including custom shaders).

Here is an example of a texture and how it's applied to the mesh. You can use the texture below as a reference for the generated mesh's UV map:





Advantages

TextMesh has a few advantages over Label3D:

- TextMesh can use a texture to modify text color on a per-side basis.
- TextMesh geometry can have actual depth to it, giving glyphs a 3D look.

- TextMesh can use custom shaders, unlike Label3D.

Limitations

There are some limitations to TextMesh:

- No built-in outline support, unlike Label3D. This can be simulated using custom shaders though.
- Only dynamic fonts are supported (`.ttf`, `.otf`, `.woff`, `.woff2`). Bitmap fonts in the `.fnt` or `.font` formats are **not** supported.
- Fonts with self-intersecting outlines will not render correctly. If you notice rendering issues on fonts downloaded from websites such as Google Fonts, try downloading the font from the font author's official website instead.
- Antialiasing the text rendering requires a full-scene antialiasing method to be enabled such as MSAA, FXAA and temporal antialiasing (TAA). If no antialiasing method is enabled, text will appear grainy, especially at a distance. See 3D antialiasing for more information.

Projected Label node (or any other Control)

There is a last solution that is more complex to set up, but provides the most flexibility: projecting a 2D node onto 3D space. This can be achieved using the return value of unproject_position method on a Camera3D node in a script's `_process()` function. This return value should then be used to set the `position` property of a Control node.

See the 3D waypoints [https://github.com/godotengine/godot-demo-projects/tree/master/3d/waypoints] demo for an example of this.

Advantages

- Any Control node can be used, including Label, RichTextLabel or even nodes such as Button. This allows for powerful formatting and GUI interaction.
- The script-based approach allows for complete freedom in positioning. For example, this makes it considerably easier to pin Controls to the screen's edges when they go off-screen (for in-game 3D markers).
- Control theming is obeyed. This allows for easier customization that globally applies to the project.

Limitations

- Projected Controls cannot be occluded by 3D geometry in any way. You can use a RayCast to fully hide the control if its target position is occluded by a collider, but this doesn't allow for partially hiding the control behind a wall.
- Changing text size depending on distance by adjusting the Control's `scale` property is possible, but it needs to be done manually. Label3D and TextMesh automatically take care of this, at the cost of less flexibility (can't set a minimum/maximum text size in pixels).
- Handling resolution and aspect ratio changes must be taken into account in the script, which can be challenging.

Should I use Label3D, TextMesh or a projected Control?

In most scenarios, Label3D is recommended as it's easier to set up and provides higher rendering quality (especially if 3D antialiasing is disabled).

For advanced use cases, TextMesh is more flexible as it allows styling the text with custom shaders. Custom shaders allow for modifying the final geometry, such as curving the text along a surface. Since the text is actual 3D geometry, the text can optionally have depth to it and can also contribute to global illumination.

If you need features such as BBCode or Control theming support, then using a projected RichTextLabel node is the only way to go.

3D rendering limitations

Introduction

Due to their focus on performance, real-time rendering engines have many limitations. Godot's renderer is no exception. To work effectively with those limitations, you need to understand them.

Texture size limits

On desktops and laptops, textures larger than 8192×8192 may not be supported on older devices. You can check your target GPU's limitations on GPUinfo.org [https://www.gpuinfo.org/].

Mobile GPUs are typically limited to 4096×4096 textures. Also, some mobile GPUs don't support repeating non-power-of-two-sized textures. Therefore, if you want your texture to display correctly on all platforms, you should avoid using textures larger than 4096×4096 and use a power of two size if the texture needs to repeat.

To limit the size of a specific texture that may be too large to render, you can set the **Process > Size Limit** import option to a value greater than `0`. This will reduce the texture's dimensions on import (preserving aspect ratio) without affecting the source file.

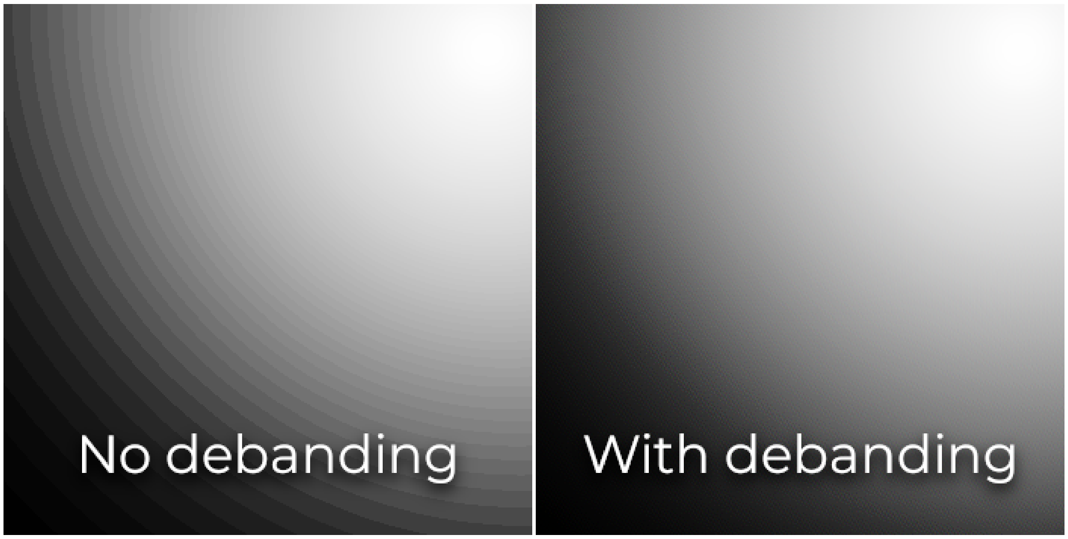
Color banding

When using the Forward+ or Forward Mobile rendering methods, Godot's 3D engine renders internally in HDR. However, the rendering output will be tonemapped to a low dynamic range so it can be displayed on the screen. This can result in visible banding, especially when using untextured materials. For performance reasons, color precision is also lower when using the Forward Mobile rendering method compared to Forward+.

When using the Compatibility rendering method, HDR is not used and the color precision is the lowest of all rendering methods. This also applies to 2D rendering, where banding may be visible when using smooth gradient textures.

There are two main ways to alleviate banding:

- If using the Forward+ or Forward Mobile rendering methods, enable **Use Debanding** in the advanced Project Settings. This applies a fullscreen debanding shader as a post-processing effect and is very cheap.
- Alternatively, bake some noise into your textures. This is mainly effective in 2D, e.g. for vignetting effects. In 3D, you can also use a custom debanding shader [https://github.com/fractilegames/godot-gles2-debanding-material] to be applied on your *materials*. This technique works even if your project is rendered with low color precision, which means it will work when using the Mobile and Compatibility rendering methods.



Color banding comparison (contrast increased for more visibility)

See also

See Banding in Games: A Noisy Rant (PDF) [https://loopit.dk/banding_in_games.pdf] for more details about banding and ways to combat it.

Depth buffer precision

To sort objects in 3D space, rendering engines rely on a *depth buffer* (also called *Z-buffer*). This buffer has a finite precision: 24-bit on desktop platforms, sometimes 16-bit on mobile platforms (for performance reasons). If two different objects end up on the same buffer value, then Z-fighting will occur. This will materialize as textures flickering back and forth as the camera moves or rotates.

To make the depth buffer more precise over the rendered area, you should *increase* the Camera node's **Near** property. However, be careful: if you set it too high, players will be able to see through nearby geometry. You should also *decrease* the Camera node's **Far** property to the lowest permissible value for your use case, though keep in mind it won't impact precision as much as the **Near** property.

If you only need high precision when the player can see far away, you could change it dynamically based on the game conditions. For instance, if the player enters an airplane, the **Near** property can be temporarily increased to avoid Z-fighting in the distance. It can then be decreased once the player leaves the airplane.

Depending on the scene and viewing conditions, you may also be able to move the Z-fighting objects further apart without the difference being visible to the player.

Label3D
Z-fighting
Z-fighting fixed
(offset Label3D)

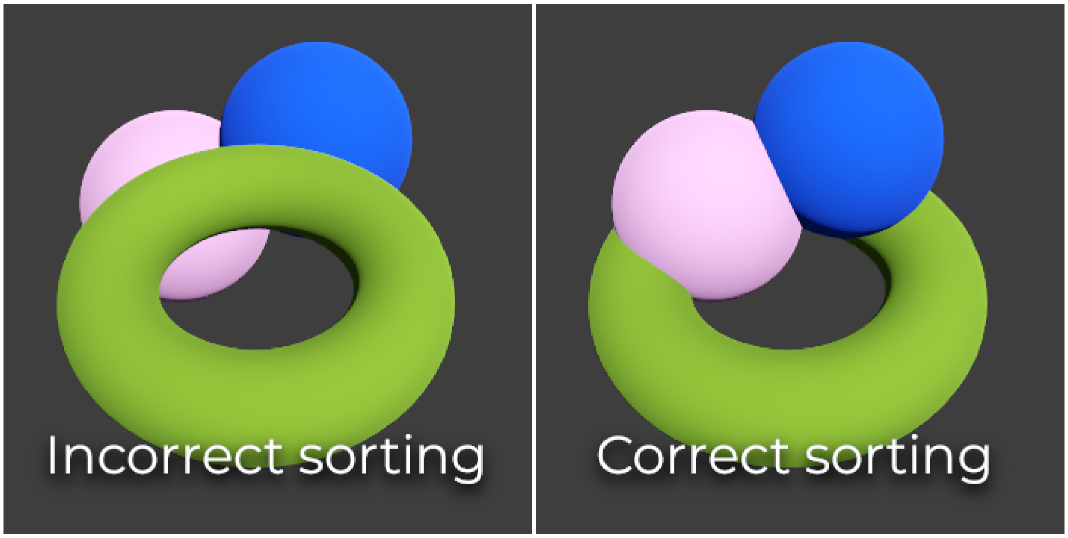
Z-fighting comparison (before and after tweaking the scene by offsetting the Label3D away from the floor)

Transparency sorting

In Godot, transparent materials are drawn after opaque materials. Transparent objects are sorted back to front before being drawn based on the Node3D's position, not the vertex position in world space. Due to this, overlapping objects may often be sorted out of order. To fix improperly sorted objects, tweak the material's Render Priority property or the node's Sorting Offset. Render Priority will force specific materials to appear in front of or behind other transparent materials, while Sorting Offset will move the object forward or backward for the purpose of sorting. Even then, these may not always be sufficient.

Some rendering engines feature *order-independent transparency* techniques to alleviate this, but this is costly on the GPU. Godot currently doesn't provide this feature. There are still several ways to avoid this problem:

- Only make materials transparent if you actually need it. If a material only has a small transparent part, consider splitting it into a separate material. This will allow the opaque part to cast shadows and will also improve performance.
- If your texture mostly has fully opaque and fully transparent areas, you can use alpha testing instead of alpha blending. This transparency mode is faster to render and doesn't suffer from transparency issues. Enable **Transparency > Transparency** to **Alpha Scissor** in StandardMaterial3D, and adjust **Transparency > Alpha Scissor Threshold** accordingly if needed. Note that MSAA will not antialias the texture's edges unless alpha antialiasing is enabled in the material's properties. However, FXAA, TAA and supersampling will be able to antialias the texture's edges regardless of whether alpha antialiasing is enabled on the material.
- If you need to render semi-transparent areas of the texture, alpha scissor isn't suitable. Instead, setting the StandardMaterial3D's **Transparency > Transparency** property to **Depth Pre-Pass** can sometimes work (at a performance cost). You can also try the **Alpha Hash** mode.
- If you want a material to fade with distance, use the StandardMaterial3D distance fade mode **Pixel Dither** or **Object Dither** instead of **Pixel Alpha**. This will make the material opaque, which also speeds up rendering.



Transparency sorting comparison (alpha-blended materials on the left, alpha scissor materials on the right)

Multi-sample antialiasing

See also

Antialiasing is explained in detail on the 3D antialiasing page.

Multi-sample antialiasing (MSAA) takes multiple *coverage* samples at the edges of polygons when rendering objects. It does not increase the number of *color* samples used to render a scene. Here's what this means in practice:

- Edges of meshes will be smoothed out nicely (as well as supersampling would).
- Transparent materials that use *alpha testing* (1-bit transparency) won't be smoothed out.
- Specular aliasing ("sparkles" that appear on reflective surfaces) won't be reduced.

There are several ways to work around this limitation depending on your performance budget:

- To make specular aliasing less noticeable, open the Project Settings and enable **Rendering > Quality > Screen Space Filters > Screen Space Roughness Limiter**. This filter has a moderate cost on performance, so it should only be enabled if you actually need it.
- Enable fast approximate antialiasing (FXAA) in addition to (or instead of) MSAA. Since FXAA is a screen-space antialiasing method, it will smooth out anything. As a downside, FXAA also makes the scene appear blurrier, especially at resolutions below 1440p. FXAA also lacks temporal information, which means its impact on specular aliasing is limited.
- Enable temporal antialiasing (TAA) in addition to (or instead of) MSAA. Since TAA is a screen-space antialiasing method, it will smooth out anything. As a downside, TAA also makes the scene appear blurrier, especially at resolutions below 1440p. TAA provides superior quality compared to FXAA and can effectively combat specular aliasing. However, TAA has a greater performance cost compared to FXAA, and TAA can introduce ghosting artifacts with fast movement.
- Render the scene at a higher resolution by increasing the **Scaling 3D > Scale** project setting above `1.0`. This technique is called supersample antialiasing (SSAA) and is very slow. Its use is generally only recommended for offline rendering.

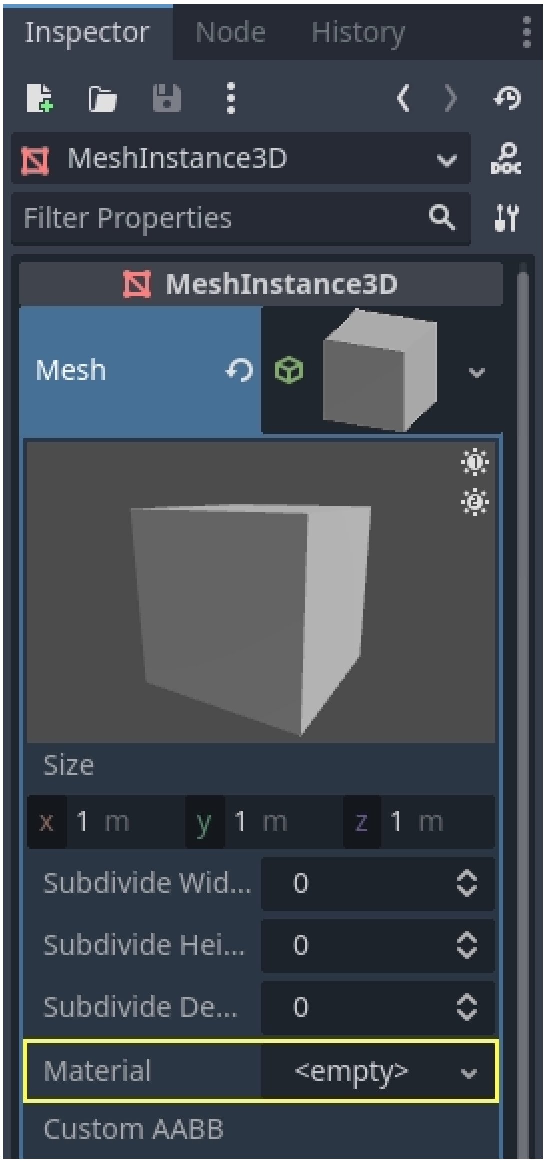
Standard Material 3D and ORM Material 3D

Introduction

`StandardMaterial3D` and `ORMMaterial3D` (Occlusion, Roughness, Metallic) are default 3D materials that aim to provide most of the features artists look for in a material, without the need for writing shader code. However, they can be converted to shader code if additional functionality is needed.

This tutorial explains the parameters present in both materials.

There are 4 ways to add these materials to an object. A material can be added in the *Material* property of the mesh. It can be added in the *Material* property of the node using the mesh (such as a MeshInstance3D node), the *Material Override* property of the node using the mesh, and the *Material Overlay*.


Inspector
Node
History
MeshInstance3D
Filter Properties
MeshInstance3D

Mesh

Size

x 1 m y 1 m z 1 m

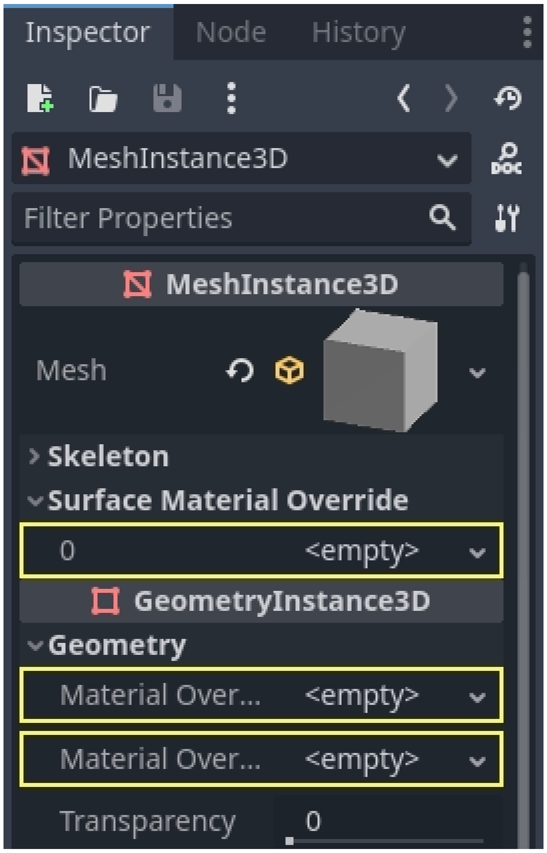
Subdivide Wid... 0

Subdivide Hei... 0

Subdivide De... 0

Material <empty>

Custom AABB


Inspector
Node
History
MeshInstance3D
Filter Properties
MeshInstance3D

Mesh

Skeleton

Surface Material Override

0 <empty>

GeometryInstance3D

Geometry

Material Over... <empty>

Material Over... <empty>

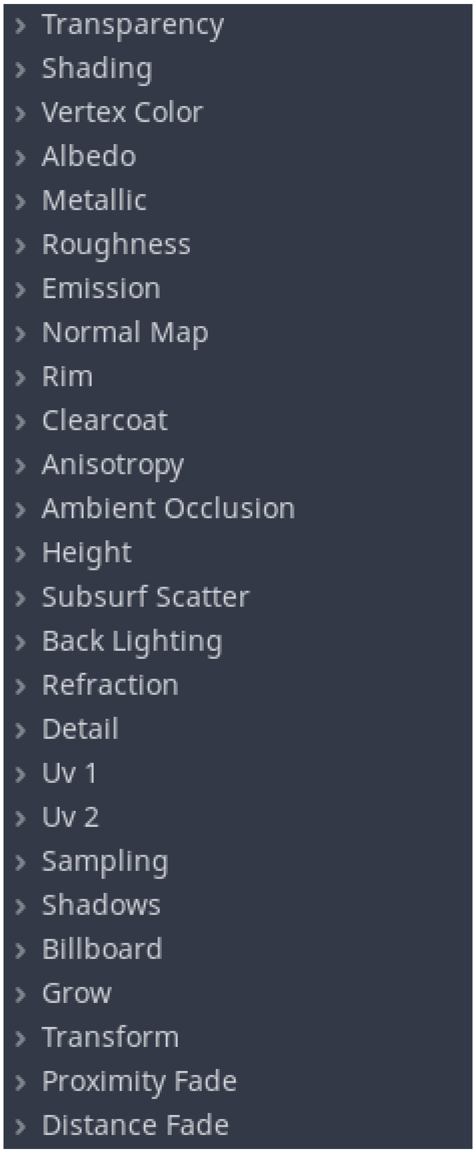
Transparency 0

If you add a material to the mesh itself, every time that mesh is used it will have that material. If you add a material to the node using the mesh, the material will only be used by that node, it will also override the material property of the mesh. If a material is added in the *Material Override* property of the node, it will only be used by that node. It will also override the regular material property of the node and the material property of the mesh.

The *Material Overlay* property will render a material **over** the current one being used by the mesh. As an example, this can be used to put a transparent shield effect on a mesh.

BaseMaterial 3D settings

StandardMaterial3D has many settings that determine the look of a material. All of these are under the BaseMaterial3D category



Transparency
Shading
Vertex Color
Albedo
Metallic
Roughness
Emission
Normal Map
Rim
Clearcoat
Anisotropy
Ambient Occlusion
Height
Subsurf Scatter
Back Lighting
Refraction
Detail
Uv 1
Uv 2
Sampling
Shadows
Billboard
Grow
Transform
Proximity Fade
Distance Fade

ORM materials are almost exactly the same with one difference. Instead of separate settings and textures for occlusion, roughness, and metallic, there is a single ORM texture. The different color channels of that texture are used for each parameter. Programs such as Substance Painter and Armor Paint will give you the option to export in this format, for these two programs it's with the export preset for unreal engine, which also uses ORM textures.

Transparency

By default, materials in Godot are opaque. This is fast to render, but it means the material can't be seen through even if you use a transparent texture in the **Albedo > Texture** property (or set **Albedo > Color** to a transparent color).

To be able to see through a material, the material needs to be made *transparent*. Godot offers several transparency modes:

- **Disabled:** Material is opaque. This is the fastest to render, with all rendering features supported.
- **Alpha:** Material is transparent. Semi-transparent areas are drawn with blending. This is slow to render, but it allows for partial transparency (also known as translucency). Materials using alpha blending also can't cast shadows, and are not visible in screen-space reflections.
    - **Alpha** is a good fit for particle effects and VFX.
- **Alpha Scissor:** Material is transparent. Semi-transparent areas whose opacity is below **Alpha Scissor Threshold** are not drawn (above this opacity, these are drawn as opaque). This is faster to render than Alpha and doesn't exhibit transparency sorting issues. The downside is that this results in "all or nothing" transparency, with no intermediate values possible. Materials using alpha scissor can cast shadows.
    - **Alpha Scissor** is ideal for foliage and fences, since these have hard edges and require correct sorting to look good.
- **Alpha Hash:** Material is transparent. Semi-transparent areas are drawn using dithering. This is also "all or nothing" transparency, but dithering helps represent partially opaque areas with limited precision depending on viewport resolution. Materials using alpha hash can cast shadows.

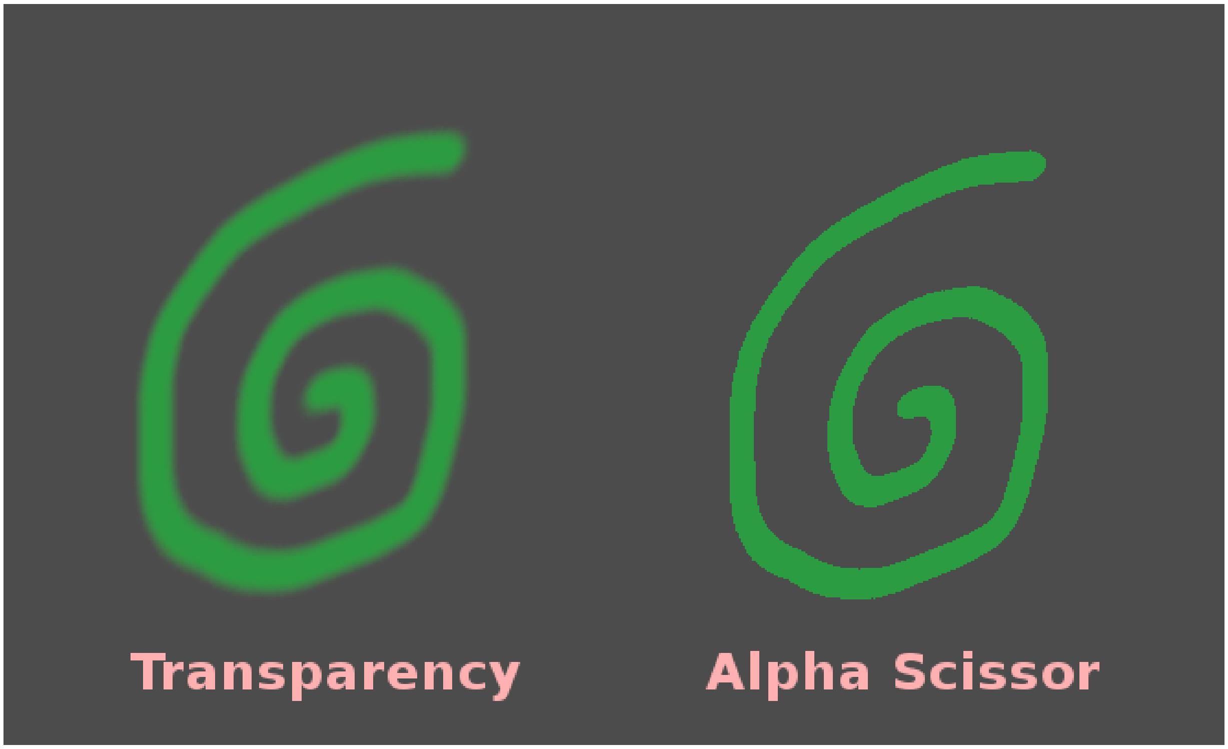
    - **Alpha Hash** is suited for realistic-looking hair, although stylized hair may work better with alpha scissor.
- **Depth Pre-Pass:** This renders the object's fully opaque pixels via the opaque pipeline first, then renders the rest with alpha blending. This allows transparency sorting to be *mostly* correct (albeit not fully so, as partially transparent regions may still exhibit incorrect sorting). Materials using depth prepass can cast shadows.

Note

Godot will automatically force the material to be transparent with alpha blending if *any* of these conditions is met:

- Setting the transparency mode to **Alpha** (as described here).
- Setting a blend mode other than the default **Mix**
- Enabling **Refraction**, **Proximity Fade**, or **Distance Fade**.

Comparison between alpha blending (left) and alpha scissor (right) transparency:



Warning

Alpha-blended transparency has several limitations:

- Alpha-blended materials are significantly slower to render, especially if they overlap.
- Alpha-blended materials may exhibit sorting issues when transparent surfaces overlap each other. This means that surfaces may render in the incorrect order, with surfaces in the back appearing to be in front of those which are actually closer to the camera.
- Alpha-blended materials don't cast shadows, although they can receive shadows.
- Alpha-blended materials don't appear in any reflections (other than reflection probes).
- Screen-space reflections and sharp SDFGI reflections don't appear on alpha-blended materials. When SDFGI is enabled, rough reflections are used as a fallback regardless of material roughness.

Before using the **Alpha** transparency mode, always consider whether another transparency mode is more suited for your needs.

Alpha Antialiasing

Note

This property is only visible when the transparency mode is **Alpha Scissor** or **Alpha Hash**.

While alpha scissor and alpha hash materials are faster to render than alpha-blended materials, they exhibit hard edges between opaque and transparent regions. While it's possible to use post-processing-based antialiasing techniques such as FXAA and TAA, this is not always desired as these techniques tend to make the final result look blurrier or exhibit ghosting artifacts.

There are 3 alpha antialiasing modes available:

- **Disabled:** No alpha antialiasing. Edges of transparent materials will appear aliased unless a post-processing-based antialiasing solution is used.
- **Alpha Edge Blend:** Results in a smooth transition between opaque and transparent areas. Also known as "alpha to coverage".
- **Alpha Edge Clip:** Results in a sharp, but still antialiased transition between opaque and transparent areas. Also known as "alpha to coverage + alpha to one".

When the alpha antialiasing mode is set to **Alpha Edge Blend** or **Alpha Edge Clip**, a new **Alpha Antialiasing Edge** property becomes visible below in the inspector. This property controls the threshold below which pixels should be made transparent. While you've already defined an alpha scissor threshold (when using **Alpha Scissor** only), this additional threshold is used to smoothly transition between opaque and transparent pixels. **Alpha Antialiasing Edge** must *always* be set to a value that is strictly below the alpha scissor threshold. The default of `0.3` is a sensible value with an alpha scissor of threshold of `0.5`, but remember to adjust this alpha antialiasing edge when modifying the alpha scissor threshold.

If you find the antialiasing effect not effective enough, try increasing **Alpha Antialiasing Edge** while making sure it's below **Alpha Scissor Threshold** (if the material uses alpha scissor). On the other hand, if you notice the texture's appearance visibly changing as the camera moves closer to the material, try decreasing **Alpha Antialiasing Edge**.

Important

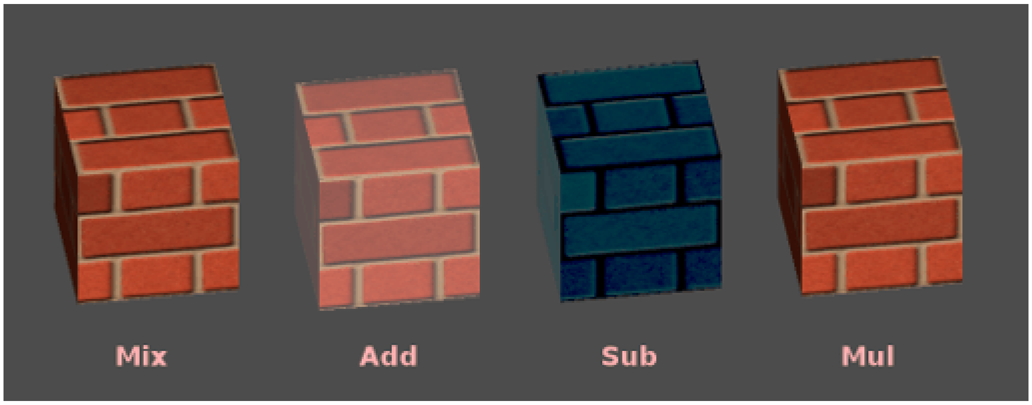
For best results, MSAA 3D should be set to at least 2× in the Project Settings when using alpha antialiasing. This is because this feature relies on alpha to coverage, which is a feature provided by MSAA.

Without MSAA, a fixed dithering pattern is applied on the material's edges, which isn't very effective at smoothing out edges (although it can still help a little).

Blend Mode

Controls the blend mode for the material. Keep in mind that any mode other than *Mix* forces the object to go through the transparent pipeline.

- **Mix:** Default blend mode, alpha controls how much the object is visible.
- **Add:** The final color of the object is added to the color of the screen, nice for flares or some fire-like effects.
- **Sub:** The final color of the object is subtracted from the color of the screen.
- **Mul:** The final color of the object is multiplied with the color of the screen.



Cull Mode

Determines which side of the object is not drawn when backfaces are rendered:

- **Back:** The back of the object is culled when not visible (default).
- **Front:** The front of the object is culled when not visible.
- **Disabled:** Used for objects that are double-sided (no culling is performed).

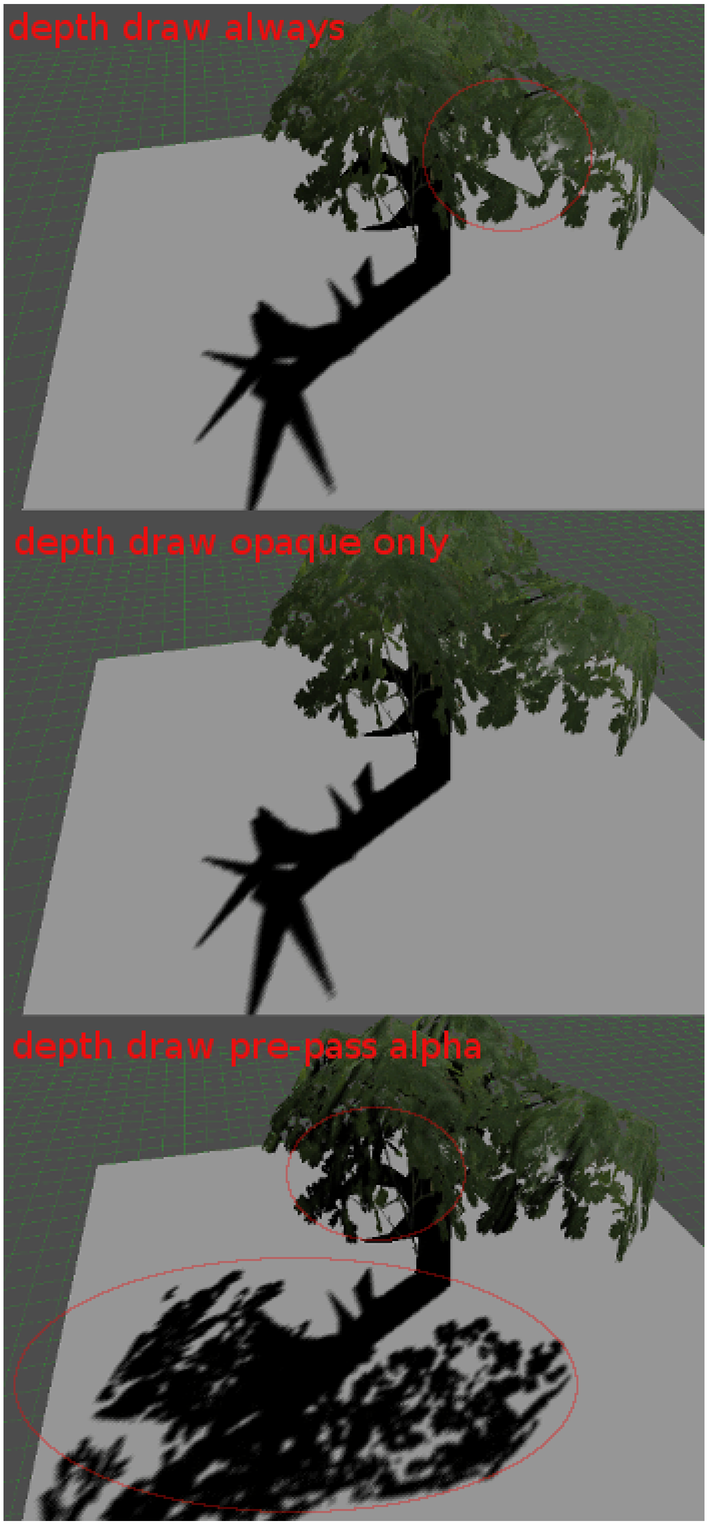
Note

By default, Blender has backface culling disabled on materials and will export materials to match how they render in Blender. This means that materials in Godot will have their cull mode set to **Disabled**. This can decrease performance since backfaces will be rendered, even when they are being culled by other faces. To resolve this, enable **Backface Culling** in Blender's Materials tab, then export the scene to glTF again.

Depth Draw Mode

Specifies when depth rendering must take place.

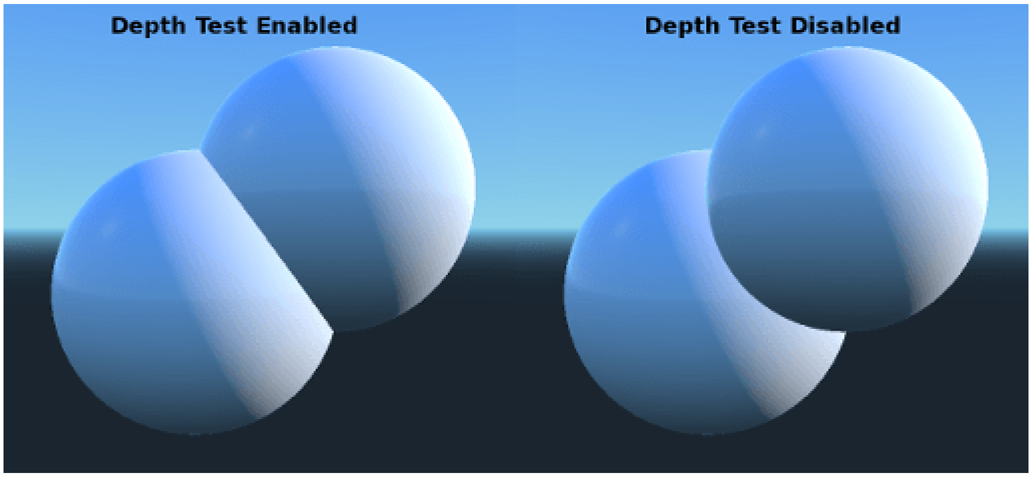
- **Opaque Only (default):** Depth is only drawn for opaque objects.
- **Always:** Depth draw is drawn for both opaque and transparent objects.
- **Never:** No depth draw takes place (do not confuse this with the No Depth Test option below).
- **Depth Pre-Pass:** For transparent objects, an opaque pass is made first with the opaque parts, then transparency is drawn above. Use this option with transparent grass or tree foliage.


depth draw always
depth draw opaque only
depth draw pre-pass alpha

No Depth Test

In order for close objects to appear over far away objects, depth testing is performed. Disabling it has the result of objects appearing over (or under) everything else.

Disabling this makes the most sense for drawing indicators in world space, and works very well with the *Render Priority* property of Material (see the bottom of this page).



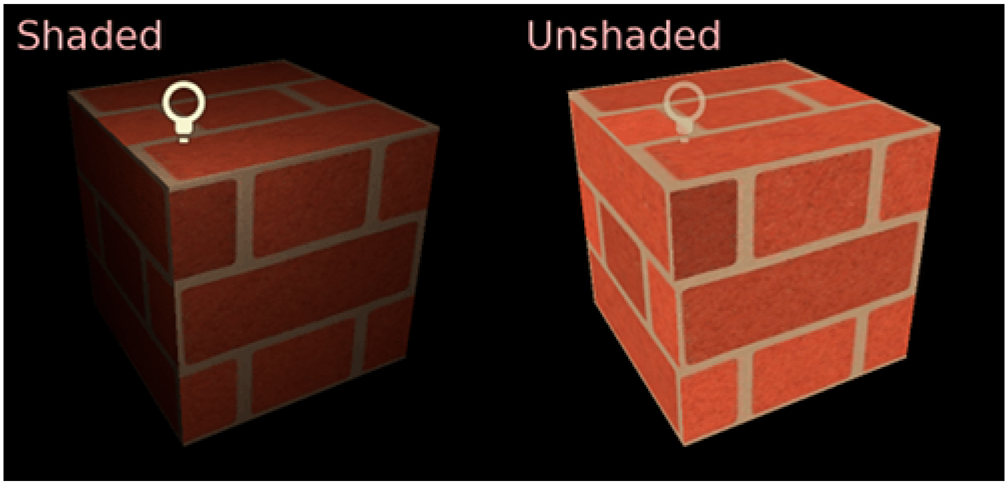
Shading

Shading mode

Godot has a more or less uniform cost per pixel thanks to the depth pre-pass. All lighting calculations are made by running the lighting shader on every pixel.

As these calculations are costly, performance can be brought down considerably in some corner cases such as drawing several layers of transparency (which is common in particle systems). Switching to the **Unshaded** shading mode may help improve performance in these cases, especially when the camera is close to particles.

Additionally, on low-end or mobile devices, switching to unshaded rendering can considerably increase rendering performance.



Keep in mind that when unshaded rendering is enabled, lights will not affect the material at all.

Note

Per-Vertex shading is listed as an option in the shading mode property. However, per-vertex shading is currently unimplemented and will act identical to per-pixel shading.

Support for per-vertex shading is planned to be reimplemented in a future Godot release.

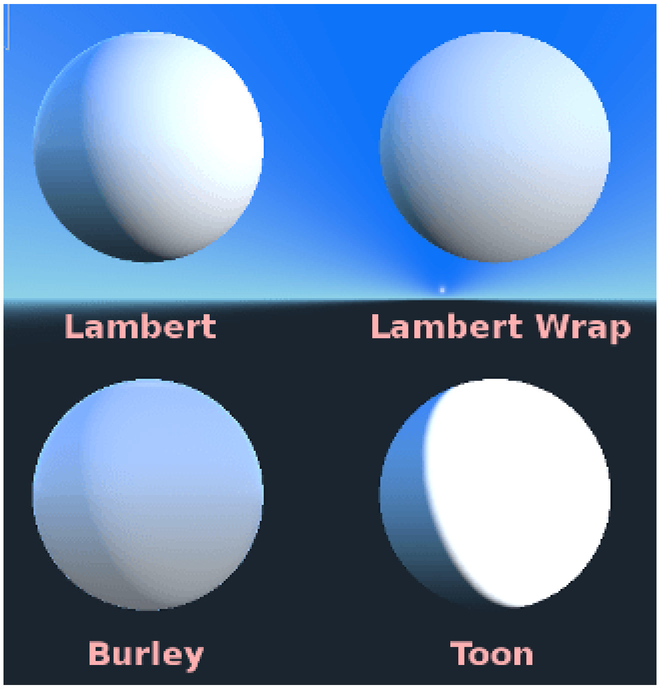
Diffuse Mode

Specifies the algorithm used by diffuse scattering of light when hitting the object. The default is **Burley**. Other modes are also available:

- **Burley:** Default mode, the original Disney Principled PBS diffuse algorithm.
- **Lambert:** Is not affected by roughness.
- **Lambert Wrap:** Extends Lambert to cover more than 90 degrees when roughness increases. Works great for hair and

simulating cheap subsurface scattering. This implementation is energy conserving.

- **Toon:** Provides a hard cut for lighting, with smoothing affected by roughness. It is recommended you disable sky contribution from your environment's ambient light settings or disable ambient light in the StandardMaterial3D to achieve a better effect.

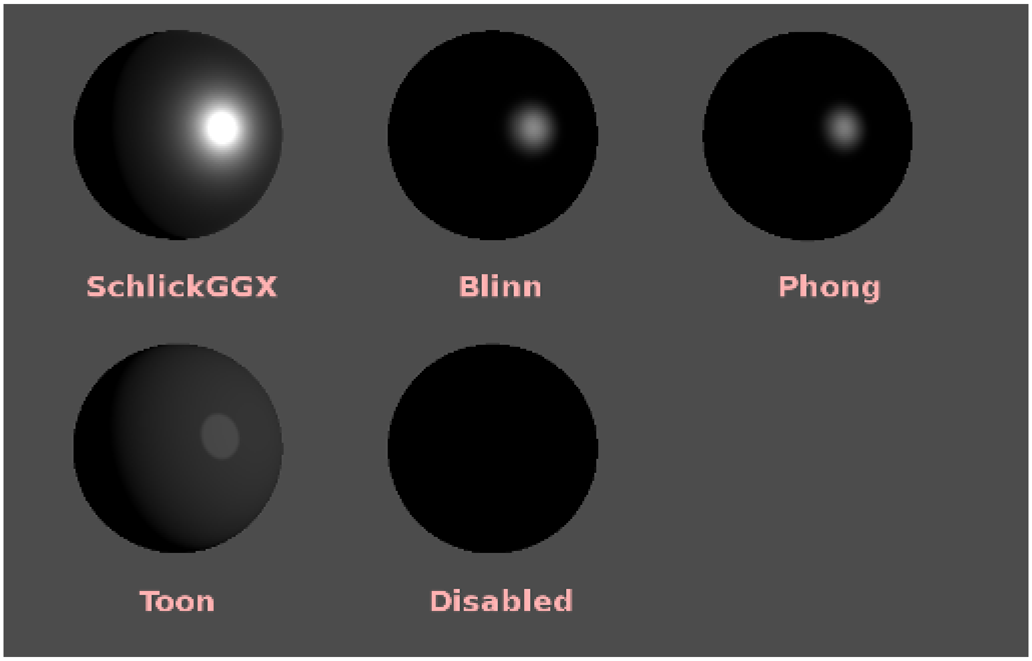


Specular Mode

Specifies how the specular blob will be rendered. The specular blob represents the shape of a light source reflected in the object.

- **SchlickGGX:** The most common blob used by PBR 3D engines nowadays.
- **Blinn:** Common in previous-generation engines. Not worth using nowadays, but left here for the sake of compatibility.
- **Phong:** Same as above.
- **Toon:** Creates a toon blob, which changes size depending on roughness.

- **Disabled:** Sometimes the blob gets in the way. Begone!



Disable Ambient Light

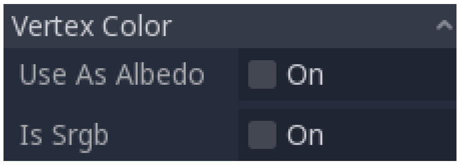
Makes the object not receive any kind of ambient lighting that would otherwise light it.

Disable Fog

Makes the object unaffected by depth-based or volumetric fog. This is useful for particles or other additively blended materials that would otherwise show the shape of the mesh (even in places where it would be invisible without the fog).

Vertex Color

This setting allows choosing what is done by default to vertex colors that come from your 3D modeling application. By default, they are ignored.



Vertex Color

Use As Albedo ☐ On

Is Srgb ☐ On

Use as Albedo

Choosing this option means vertex color is used as albedo color.

Is sRGB

Most 3D modeling software will likely export vertex colors as sRGB, so toggling this option on will help them look correct.

Albedo

Albedo is the base color for the material, on which all the other settings operate. When set to *Unshaded*, this is the only color that is visible. In previous versions of Godot, this channel was named *Diffuse*. The change of name mainly happened because, in PBR (Physically Based Rendering), this color affects many more calculations than just the diffuse lighting path.

Albedo color and texture can be used together as they are multiplied.

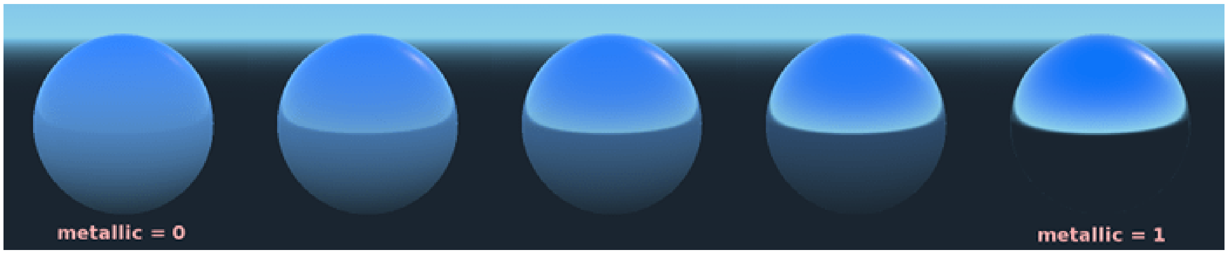
Alpha channel in albedo color and texture is also used for the object transparency. If you use a color or texture with *alpha channel*, make sure to either enable transparency or *alpha scissoring* for it to work.

Metallic

Godot uses a metallic model over competing models due to its simplicity. This parameter defines how reflective the material is. The more reflective, the less diffuse/ambient light affects the material and the more light is reflected. This model is called "energy-conserving".

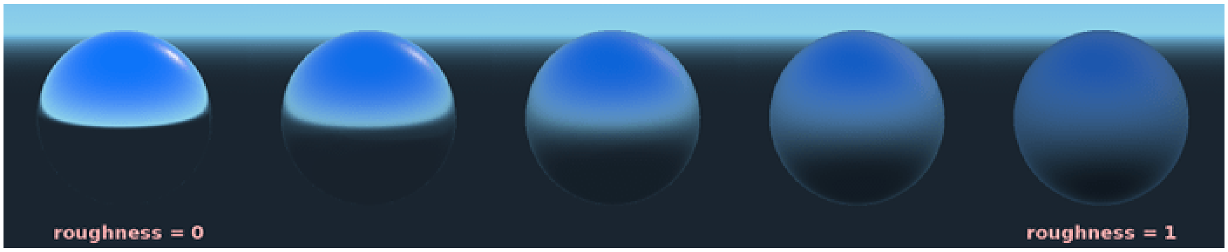
The *Specular* parameter is a general amount for the reflectivity (unlike *Metallic*, this is not energy-conserving, so leave it at `0.5` and don't touch it unless you need to).

The minimum internal reflectivity is `0.04`, so it's impossible to make a material completely unreflective, just like in real life.



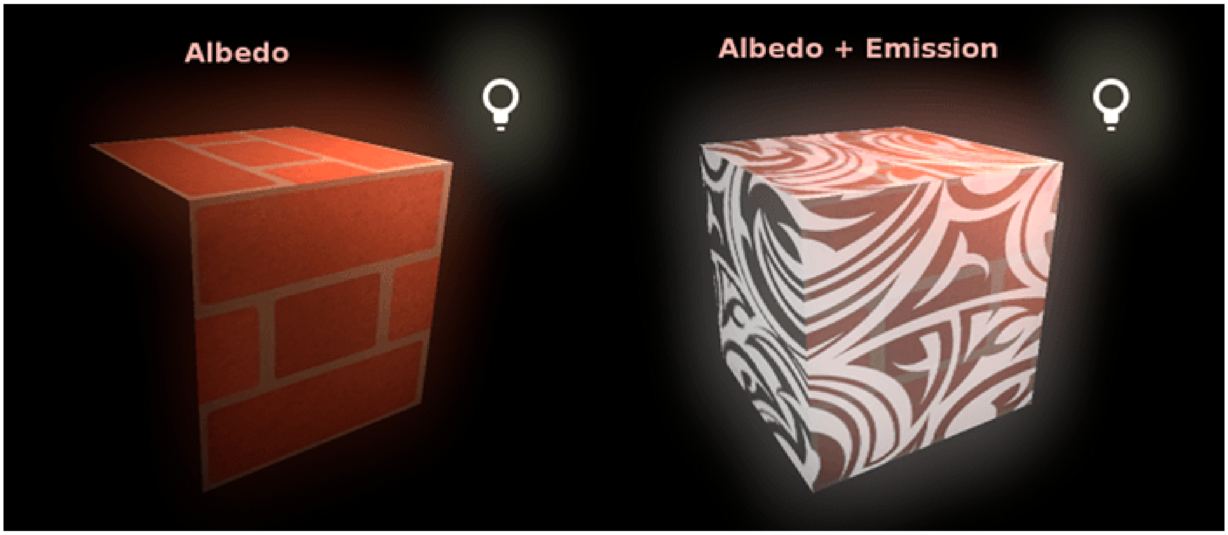
Roughness

Roughness affects the way reflection happens. A value of `0` makes it a perfect mirror while a value of `1` completely blurs the reflection (simulating natural microsurfacing). Most common types of materials can be achieved with the right combination of *Metallic* and *Roughness*.



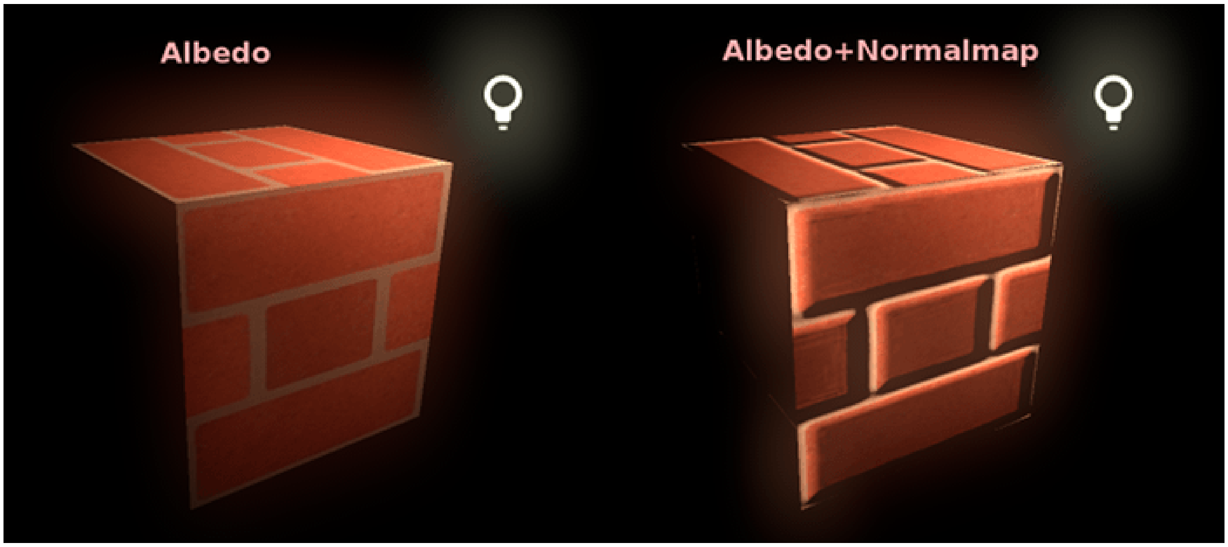
Emission

Emission specifies how much light is emitted by the material (keep in mind this does not include light surrounding geometry unless VoxelGI or SDFGI are used). This value is added to the resulting final image and is not affected by other lighting in the scene.



Normal map

Normal mapping allows you to set a texture that represents finer shape detail. This does not modify geometry, only the incident angle for light. In Godot, only the red and green channels of normal maps are used for better compression and wider compatibility.



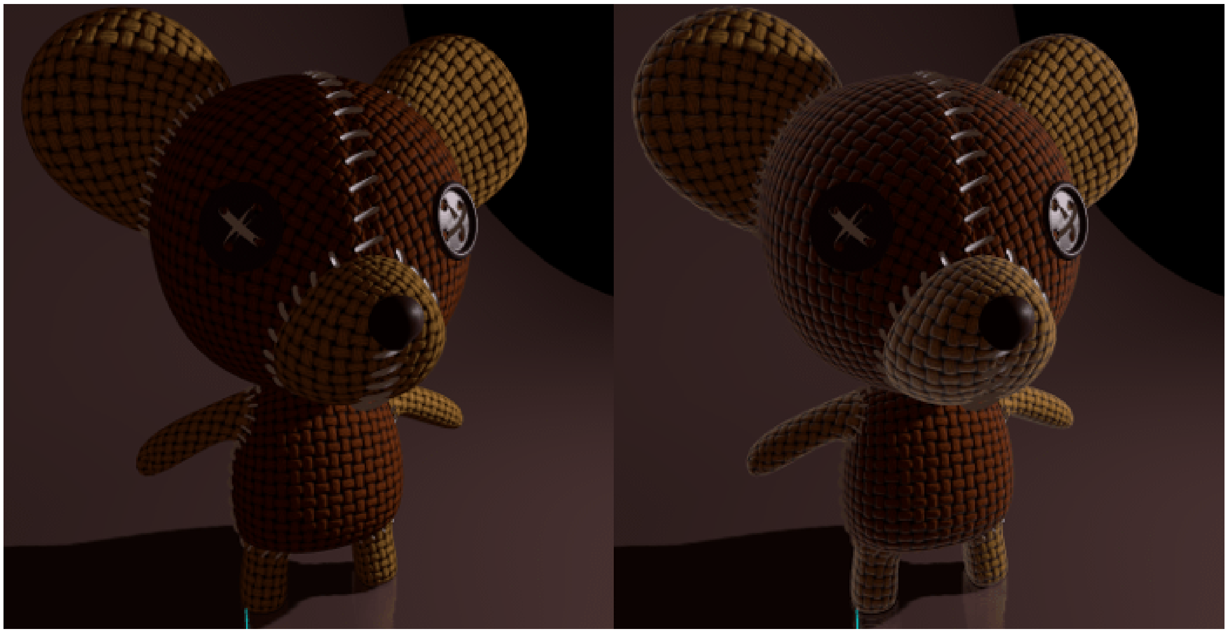
Note

Godot requires the normal map to use the X+, Y+ and Z+ coordinates, this is known as OpenGL style. If you've imported a material made to be used with another engine it may be DirectX style, in which case the normal map needs to be converted so its Y axis is flipped.

More information about normal maps (including a coordinate order table for popular engines) can be found here [http://wiki.polycount.com/wiki/Normal_Map_Technical_Details].

Rim

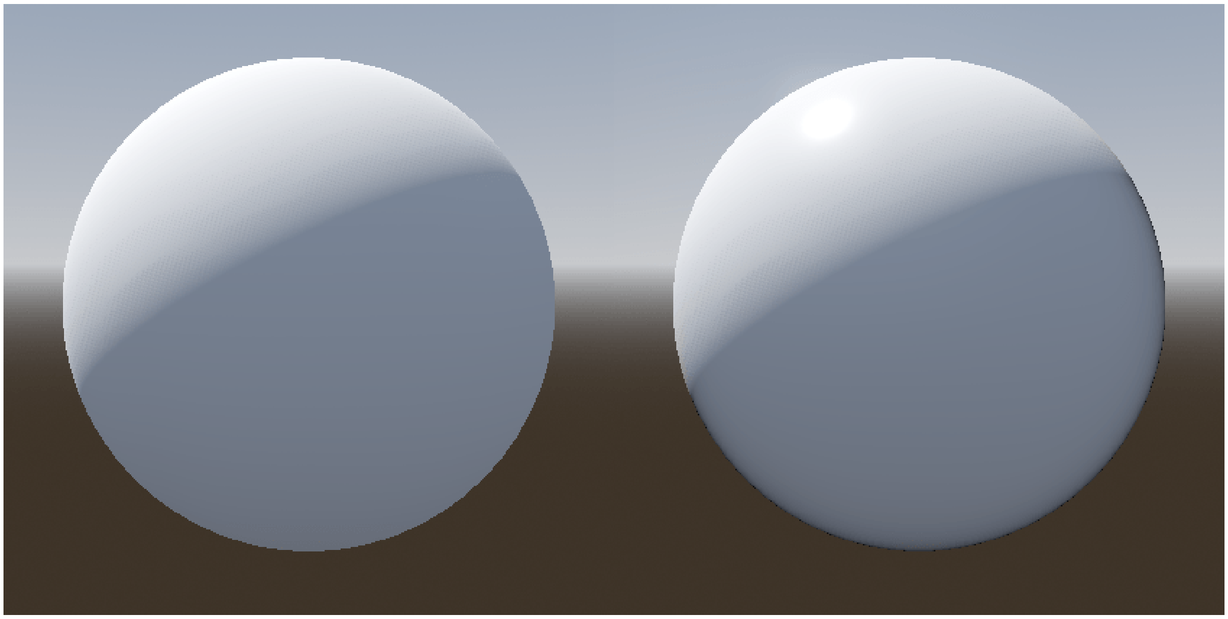
Some fabrics have small micro-fur that causes light to scatter around it. Godot emulates this with the *Rim* parameter. Unlike other rim lighting implementations, which just use the emission channel, this one actually takes light into account (no light means no rim). This makes the effect considerably more believable.



Rim size depends on roughness, and there is a special parameter to specify how it must be colored. If *Tint* is `0`, the color of the light is used for the rim. If *Tint* is `1`, then the albedo of the material is used. Using intermediate values generally works best.

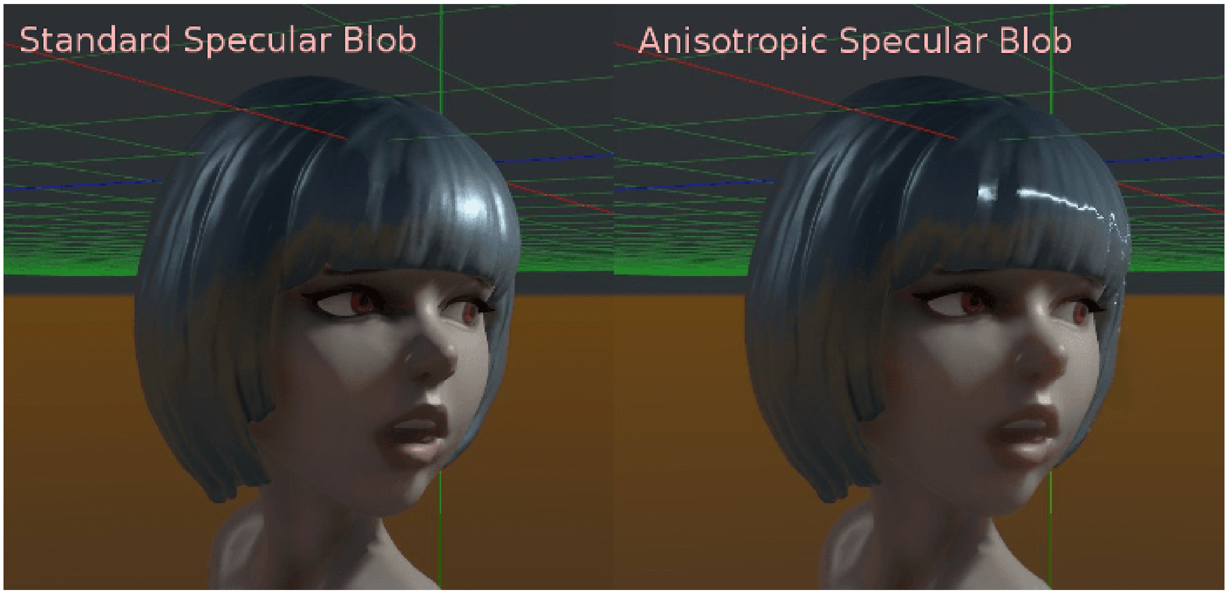
Clearcoat

The *Clearcoat* parameter is used to add a secondary pass of transparent coat to the material. This is common in car paint and toys. In practice, it's a smaller specular blob added on top of the existing material.



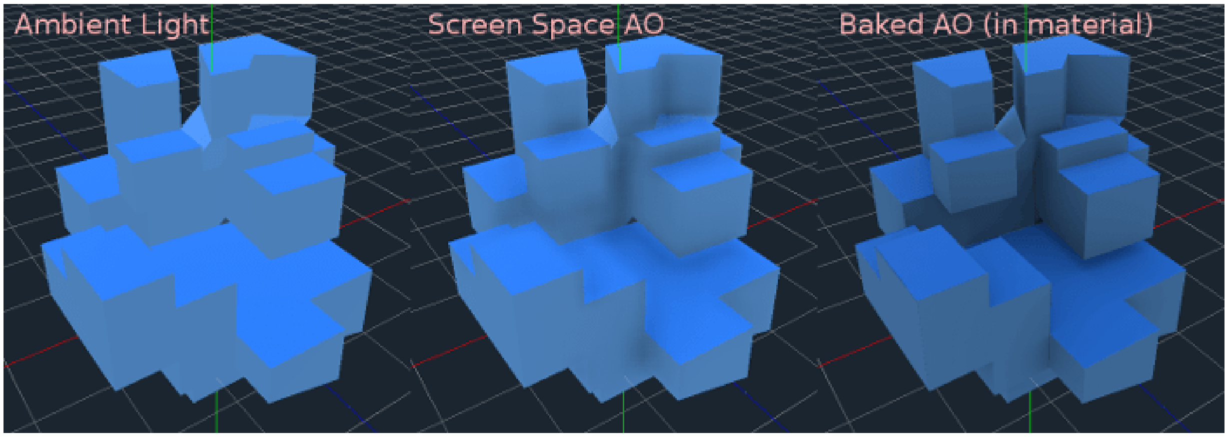
Anisotropy

This changes the shape of the specular blob and aligns it to tangent space. Anisotropy is commonly used with hair, or to make materials such as brushed aluminum more realistic. It works especially well when combined with flowmaps.


Standard Specular Blob
Anisotropic Specular Blob

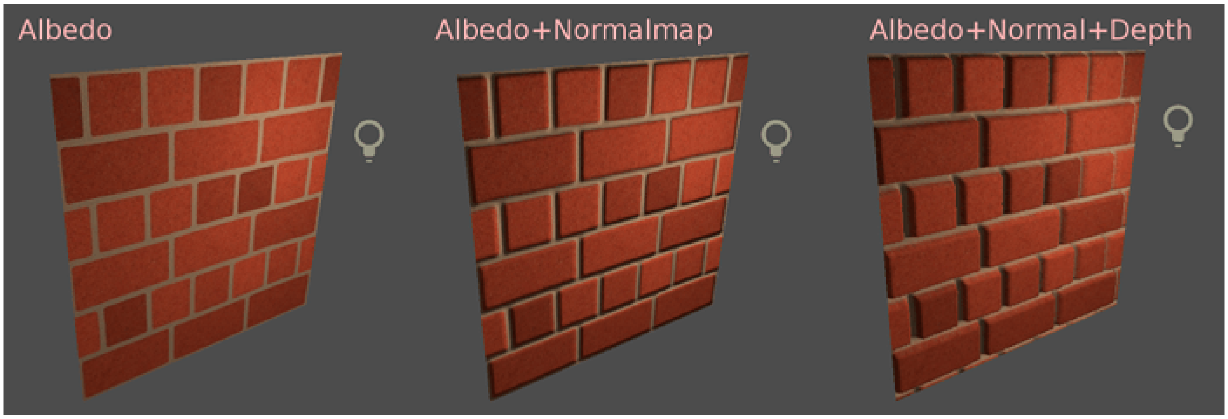
Ambient Occlusion

It is possible to specify a baked ambient occlusion map. This map affects how much ambient light reaches each surface of the object (it does not affect direct light by default). While it is possible to use Screen-Space Ambient Occlusion (SSAO) to generate ambient occlusion, nothing beats the quality of a well-baked AO map. It is recommended to bake ambient occlusion whenever possible.



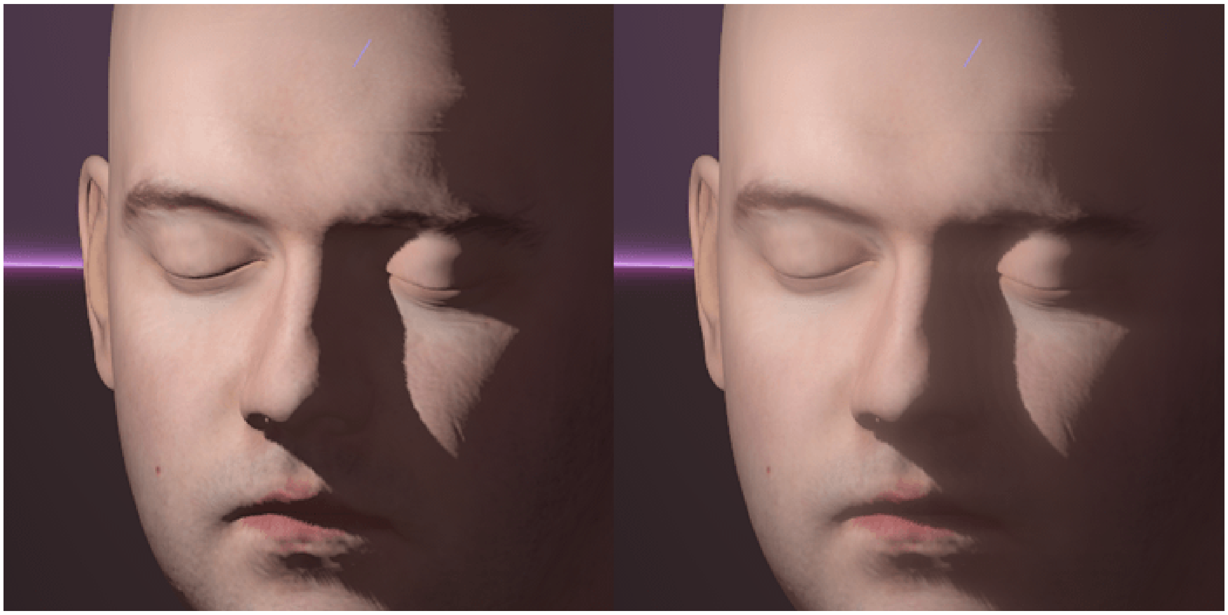
Height

Setting a height map on a material produces a ray-marched search to emulate the proper displacement of cavities along the view direction. This only creates an illusion of depth, and does not add real geometry — for a height map shape used for physics collision (such as terrain), see HeightMapShape3D. It may not work for complex objects, but it produces a realistic depth effect for textures. For best results, *Height* should be used together with normal mapping.


Albedo
Albedo+Normalmap
Albedo+Normal+Depth

Subsurface Scattering

This effect emulates light that penetrates an object's surface, is scattered, and then comes out. It is useful to create realistic skin, marble, colored liquids, etc.



Back Lighting

This controls how much light from the lit side (visible to light) is transferred to the dark side (opposite from the light). This works well for thin objects such as plant leaves, grass, human ears, etc.

Refraction

When refraction is enabled, Godot attempts to fetch information from behind the object being rendered. This allows distorting the transparency in a way similar to refraction in real life.

Remember to use a transparent albedo texture (or reduce the albedo color's alpha channel) to make refraction visible, as refraction relies on transparency to have a visible effect.

Refraction also takes the material roughness into account. Higher roughness values will make the objects behind the refraction look blurrier, which simulates real life behavior. If you can't see behind the object when refraction is enabled and albedo transparency is reduced, decrease the material's **Roughness** value.

A normal map can optionally be specified in the **Refraction Texture** property to allow distorting the refraction's direction on a per-pixel basis.

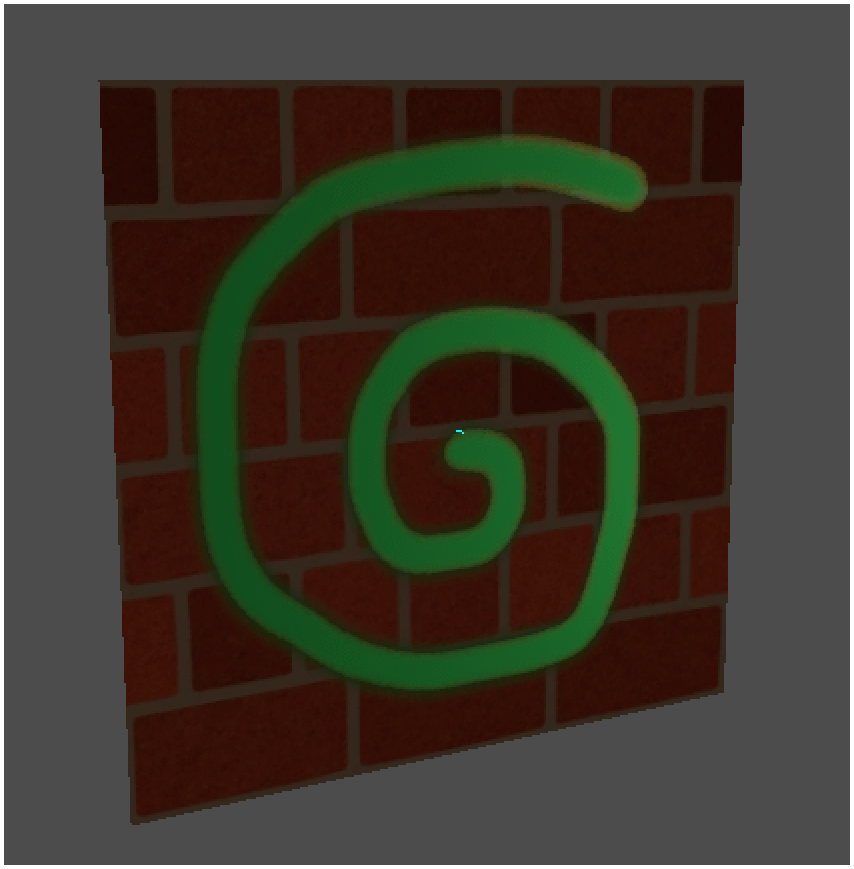
Note

Refraction is implemented as a screen-space effect and forces the material to be transparent. This makes the effect relatively fast, but this results in some limitations:

- Transparency sorting issues may occur.
- The refractive material cannot refract onto itself, or onto other transparent materials. A refractive material behind another transparent material will be invisible.
- Off-screen objects cannot appear in the refraction. This is most noticeable with high refraction strength values.
- Opaque materials in front of the refractive material will appear to have "refracted" edges, even though they shouldn't.

Detail

Godot allows using secondary albedo and normal maps to generate a detail texture, which can be blended in many ways. By combining this with secondary UV or triplanar modes, many interesting textures can be achieved.



There are several settings that control how detail is used.

Mask: The detail mask is a black and white image used to control where the blending takes place on a texture. White is for the detail textures, Black is for the regular material textures, different shades of gray are for partial blending of the material textures and detail textures.

Blend Mode: These four modes control how the textures are blended together.

- Mix: Combines pixel values of both textures. At black, only show the material texture, at white, only show the detail texture. Values of gray create a smooth blend between the two.
- Add: Adds pixel values of one Texture with the other. Unlike mix mode both textures are completely mixed at white parts of a

mask and not at gray parts. The original texture is mostly unchanged at black
- Sub: Subtracts pixel values of one texture with the other. The second texture is completely subtracted at white parts of a mask with only a little subtraction in black parts, gray parts being different levels of subtraction based on the exact texture.
- Mul: Multiplies the RGB channel numbers for each pixel from the top texture with the values for the corresponding pixel from the bottom texture.

Albedo: This is where you put an albedo texture you want to blend. If nothing is in this slot it will be interpreted as white by default.

Normal: This is where you put a normal texture you want to blend. If nothing is in this slot it will be interpreted as a flat normal map. This can still be used even if the material does not have normal map enabled.

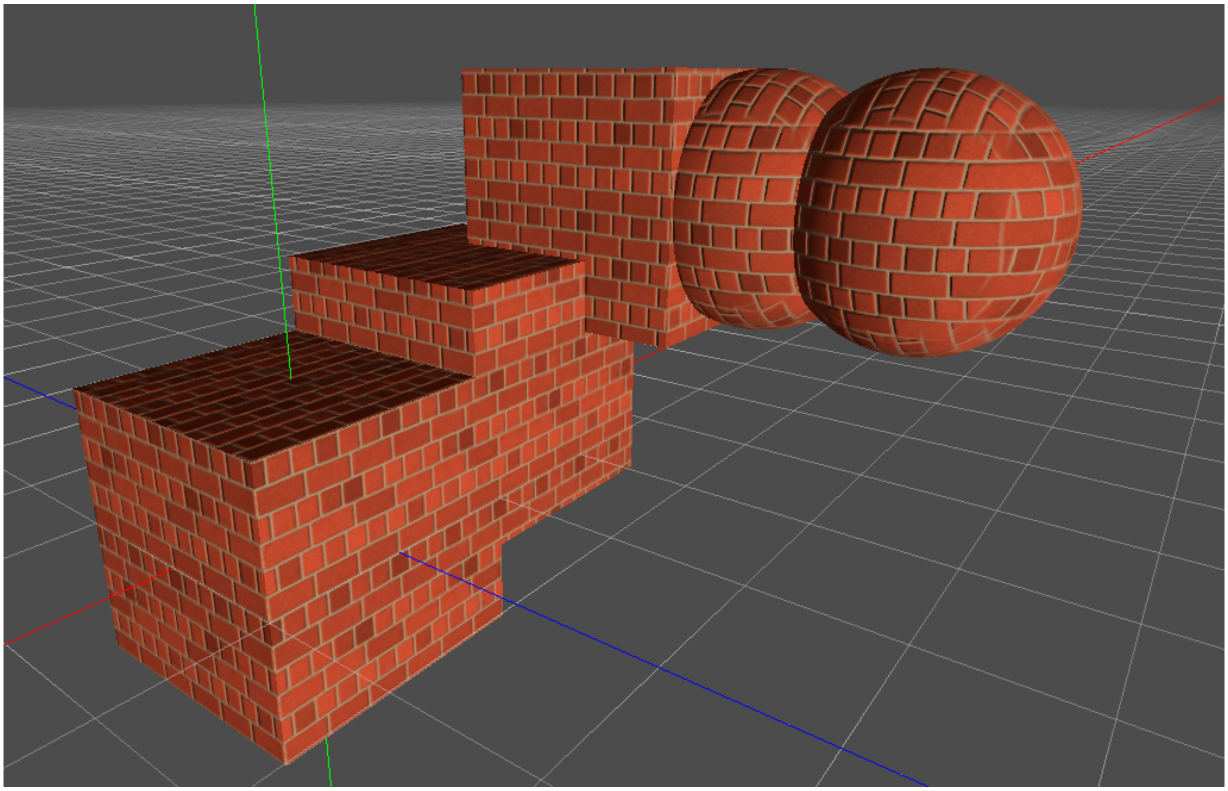
UV1 and UV2

Godot supports two UV channels per material. Secondary UV is often useful for ambient occlusion or emission (baked light). UVs can be scaled and offset, which is useful when using repeating textures.

Triplanar Mapping

Triplanar mapping is supported for both UV1 and UV2. This is an alternative way to obtain texture coordinates, sometimes called "Autotexture". Textures are sampled in X, Y and Z and blended by the normal. Triplanar mapping can be performed in either world space or object space.

In the image below, you can see how all primitives share the same material with world triplanar, so the brick texture continues smoothly between them.



World Triplanar

When using triplanar mapping, it is computed in object local space. This option makes it use world space instead.

Sampling

Filter

The filtering method for the textures used by the material. See this page for a full list of options and their description.

Repeat

if the textures used by the material repeat, and how they repeat. See this page for a full list of options and their description.

Shadows

Do Not Receive Shadows

Makes the object not receive any kind of shadow that would otherwise be cast onto it.

Use Shadow to Opacity

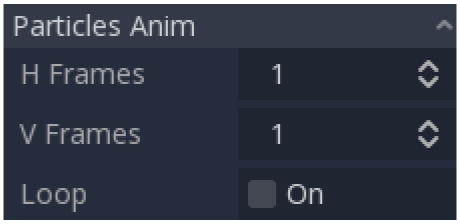
Lighting modifies the alpha so shadowed areas are opaque and non-shadowed areas are transparent. Useful for overlaying shadows onto a camera feed in AR.

Billboard

Billboard Mode

Enables billboard mode for drawing materials. This controls how the object faces the camera:

- **Disabled:** Billboard mode is disabled.
- **Enabled:** Billboard mode is enabled. The object's -Z axis will always face the camera's viewing plane.
- **Y-Billboard:** The object's X axis will always be aligned with the camera's viewing plane.
- **Particle Billboard:** Most suited for particle systems, because it allows specifying flipbook animation.


Particles Anim

H Frames 1

V Frames 1

Loop ☐ On

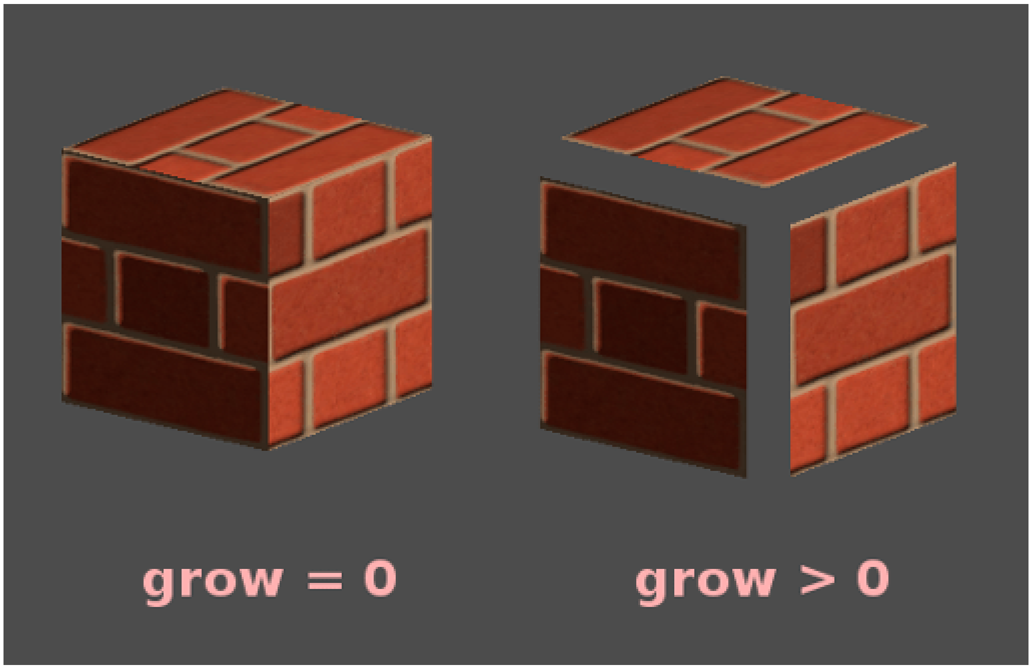
The **Particles Anim** section is only visible when the billboard mode is **Particle Billboard**.

Billboard Keep Scale

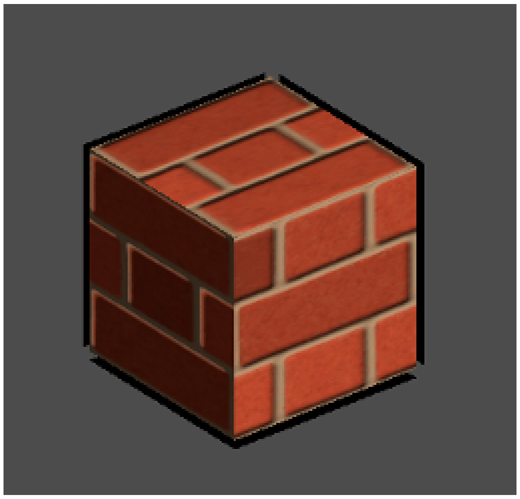
Enables scaling a mesh in billboard mode.

Grow

Grows the object vertices in the direction pointed by their normals:



This is commonly used to create cheap outlines. Add a second material pass, make it black and unshaded, reverse culling (Cull Front), and add some grow:



Transform

Fixed Size

This causes the object to be rendered at the same size no matter the distance. This is useful mostly for indicators (no depth test and high render priority) and some types of billboards.

Use Point Size

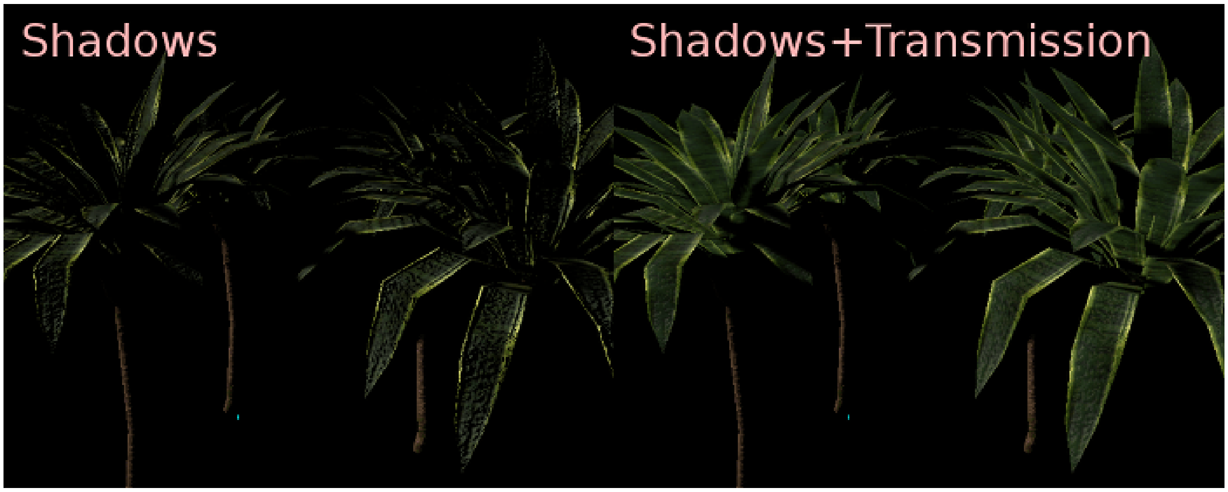
This option is only effective when the geometry rendered is made of points (generally it's made of triangles when imported from 3D modeling software). If so, then those points can be resized (see below).

Point Size

When drawing points, specify the point size in pixels.

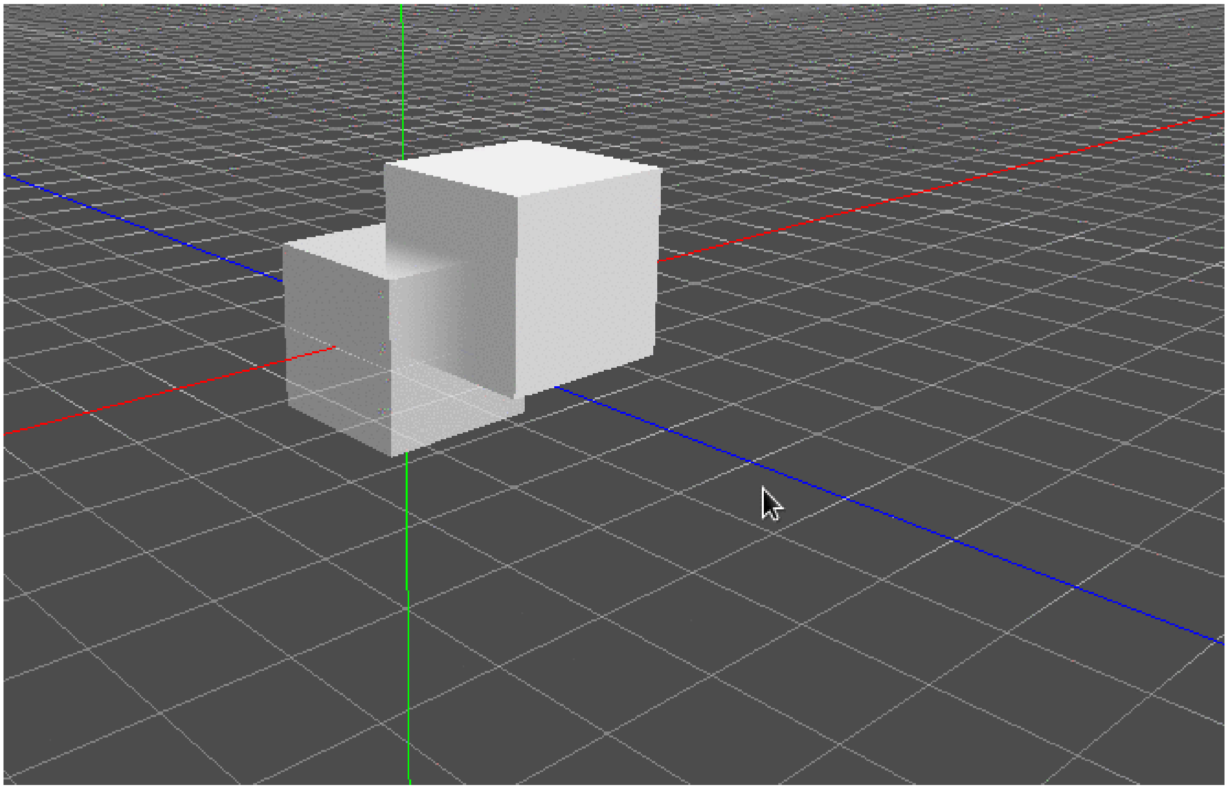
Transmission

This controls how much light from the lit side (visible to light) is transferred to the dark side (opposite from the light). This works well for thin objects such as plant leaves, grass, human ears, etc.



Proximity and Distance Fade

Godot allows materials to fade by proximity to each other as well as depending on the distance from the viewer. Proximity fade is useful for effects such as soft particles or a mass of water with a smooth blending to the shores.



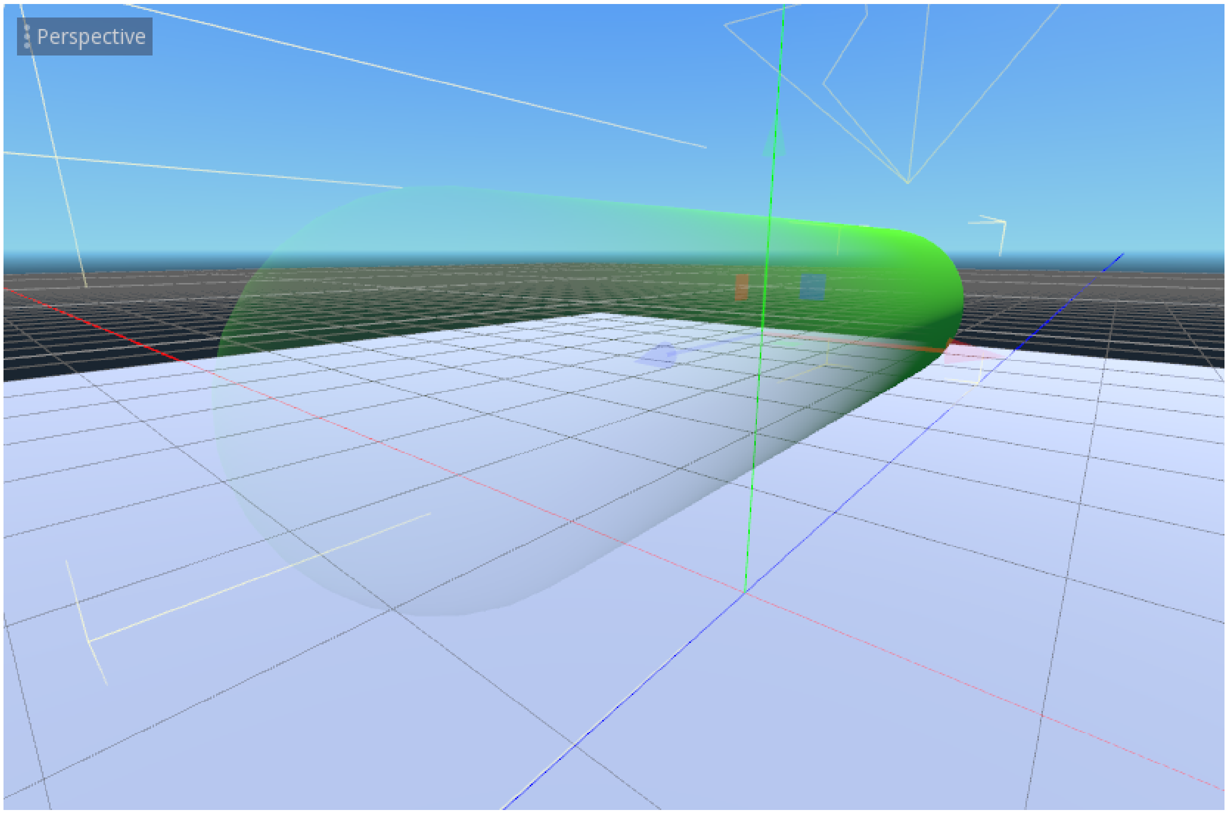
Distance fade is useful for light shafts or indicators that are only present after a given distance.

Keep in mind enabling proximity fade or distance fade with **Pixel Alpha** mode enables alpha blending. Alpha blending is more GPU-intensive and can cause transparency sorting issues. Alpha blending also disables many material features such as the ability to cast shadows.

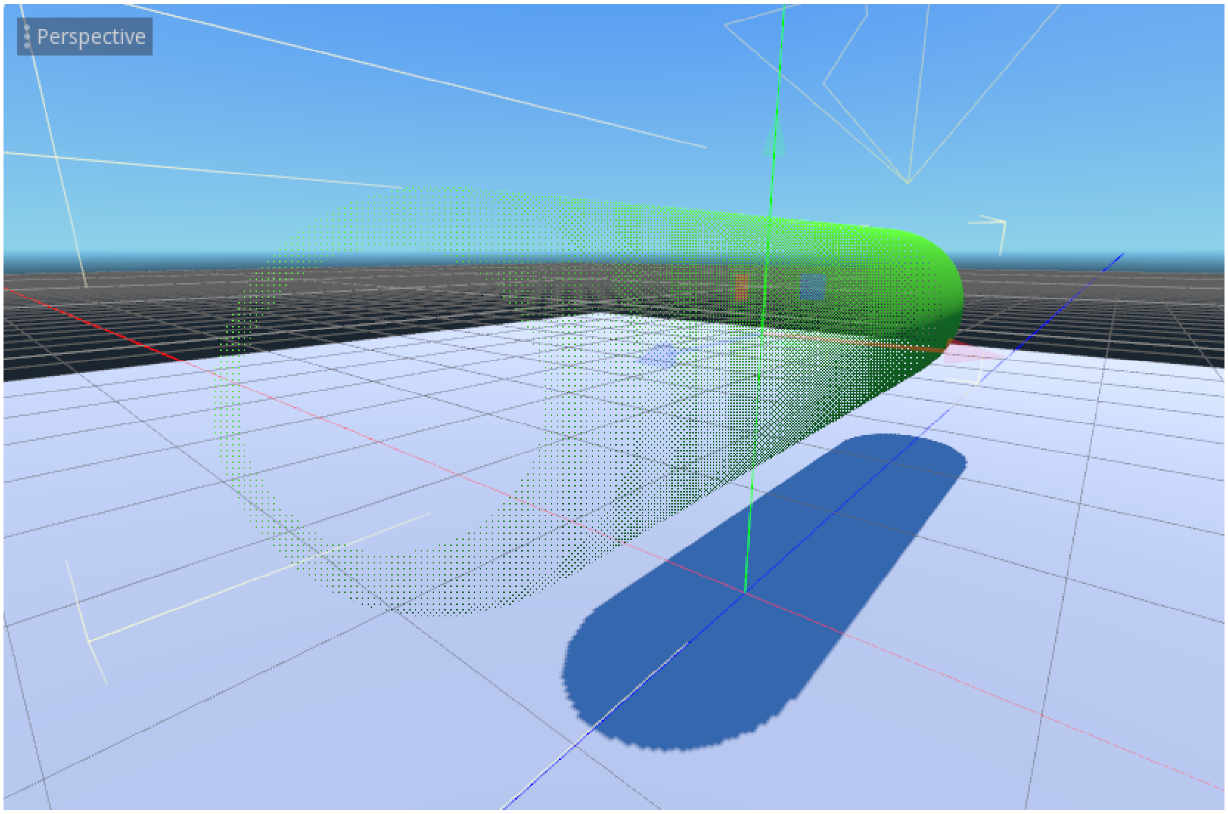
Note

To hide a character when they get too close to the camera, consider using **Pixel Dither** or better, **Object Dither** (which is even faster than **Pixel Dither**).

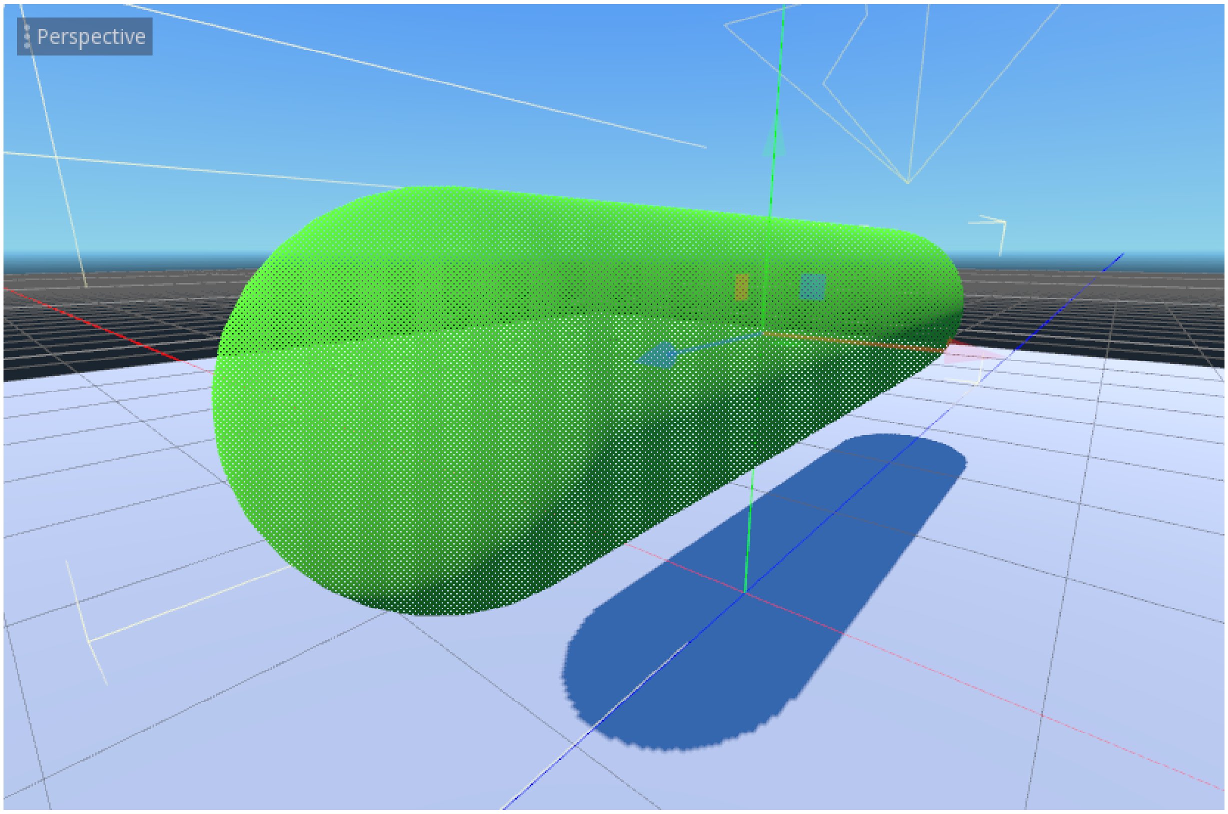
Pixel Alpha mode: The actual transparency of a pixel of the object changes with distance to the camera. This is the most effect, but forces the material into the transparency pipeline (which leads, for example, to no shadows).



Pixel Dither mode: What this does is sort of approximate the transparency by only having a fraction of the pixels rendered.



Object Dither mode: Like the previous mode, but the calculated transparency is the same across the entire object's surface.



Material Settings

Render priority

The rendering order of objects can be changed, although this is mostly useful for transparent objects (or opaque objects that perform depth draw but no color draw, such as cracks on the floor).

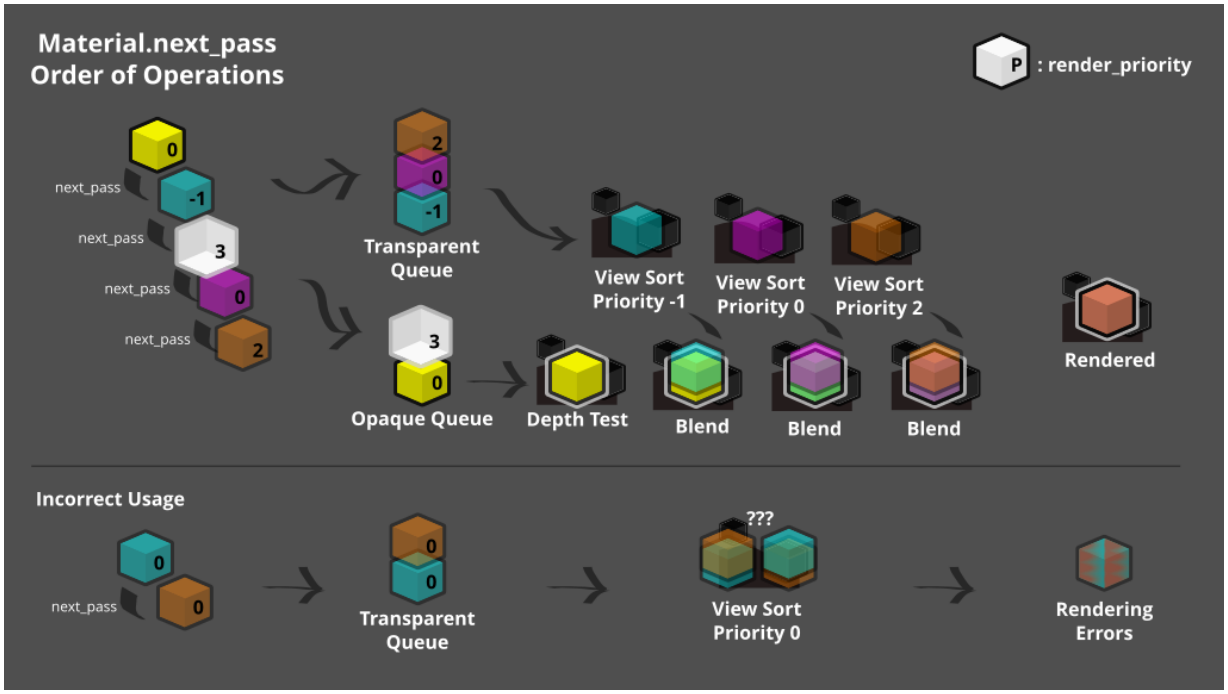
Objects are sorted by an opaque/transparent queue, then render_priority, with higher priority being drawn later. Transparent objects are also sorted by depth.

Depth testing overrules priority. Priority alone cannot force opaque objects to be drawn over each other.

Next Pass

Setting next_pass on a material will cause an object to be rendered again with that next material.

Materials are sorted by an opaque/transparent queue, then render_priority, with higher priority being drawn later.


Material.next_pass
Order of Operations
P : render_priority
0
next_pass
-1
next_pass
3
next_pass
0
next_pass
2
2
0
-1
Transparent Queue
3
0
Opaque Queue
View Sort Priority -1
View Sort Priority 0
View Sort Priority 2
Depth Test
Blend
Blend
Blend
Rendered
Incorrect Usage
0
next_pass
0
0
0
Transparent Queue
???
View Sort Priority 0
Rendering Errors

Depth will test equal between both materials unless the grow setting or other vertex transformations are used. Multiple transparent passes should use render_priority to ensure correct ordering.

3D lights and shadows

Introduction

Light sources emit light that mixes with the materials and produces a visible result. Light can come from several types of sources in a scene:

- From the material itself, in the form of the emission color (though it does not affect nearby objects unless baked or screen-space indirect lighting is enabled).
- Light nodes: DirectionalLight3D, OmniLight3D and SpotLight3D.
- Ambient light in the Environment or Reflection probes.
- Global illumination (LightmapGI, VoxelGI or SDFGI).

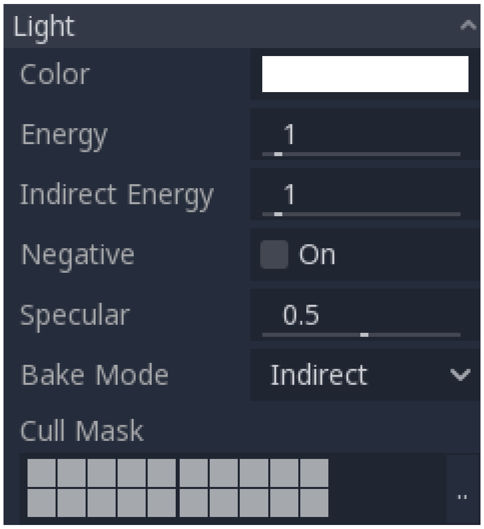
The emission color is a material property. You can read more about it in the Standard Material 3D and ORM Material 3D tutorial.

See also

You can compare various types of lights in action using the 3D Lights and Shadows demo project [https://github.com/godotengine/godot-demo-projects/tree/master/3d/lights_and_shadows].

Light nodes

There are three types of light nodes: DirectionalLight3D, OmniLight3D and SpotLight3D. Let's take a look at the common parameters for lights:



Color

Energy 1

Indirect Energy 1

Negative ☐ On

Specular 0.5

Bake Mode Indirect

Cull Mask

Each property has a specific function:

- **Color:** Base color for emitted light.
- **Energy:** Energy multiplier. This is useful for saturating lights or working with High dynamic range lighting.
- **Indirect Energy:** Secondary multiplier used with indirect light (light bounces). This works with Using Lightmap global illumination, VoxelGI or SDFGI.
- **Volumetric Fog Energy:** Secondary multiplier used with volumetric fog. This only has an effect when volumetric fog is enabled.
- **Negative:** Light becomes subtractive instead of additive. It's sometimes useful to manually compensate some dark corners.
- **Specular:** Affects the intensity of the specular blob in objects affected by this light. At zero, this light becomes a pure diffuse light.
- **Bake Mode:** Sets the bake mode for the light. See Using Lightmap global illumination.
- **Cull Mask:** Objects that are in the selected layers below will be affected by this light. Note that objects disabled via this cull mask will still cast shadows. If you don't want disabled objects to cast shadows, adjust the **Cast Shadow** property on the GeometryInstance3D to the desired value.

See also

See Physical light and camera units if you wish to use real world units to configure your lights' intensity and color temperature.

Light number limits

When using the Forward+ renderer, Godot uses a *clustering* approach for real-time lighting. As many lights as desired can be added (as long as performance allows). However, there's still a default limit of 512 *clustered elements* that can be present in the current camera view. A clustered element is an omni light, a spot light, a decal or a reflection probe. This limit can be increased by adjusting the **Rendering > Limits > Cluster Builder > Max Clustered Elements** advanced project setting.

When using the Forward Mobile renderer, there is a limitation of 8 OmniLights + 8 SpotLights per mesh resource. There is also a limit of 256 OmniLights + 256 SpotLights that can be rendered in the current camera view. These limits currently cannot be changed.

When using the Compatibility renderer, up to 8 OmniLights + 8 SpotLights can be rendered per mesh resource. This limit can be increased in the advanced Project Settings by adjusting **Rendering > Limits > OpenGL > Max Renderable Lights** and/or **Rendering > Limits > OpenGL > Max Lights Per Object** at the cost of performance and longer shader compilation times. The limit can also be decreased to reduce shader compilation times and improve performance slightly.

With all rendering methods, up to 8 DirectionalLights can be visible at a time. However, each additional DirectionalLight with shadows enabled will reduce the effective shadow resolution of each

DirectionalLight. This is because directional shadow atlas is shared between all lights.

If the rendering limit is exceeded, lights will start popping in and out during camera movement, which can be distracting. Enabling **Distance Fade** on light nodes can help reduce this issue while also improving performance. Splitting your meshes into smaller portions can also help, especially for level geometry (which also improves culling efficiency).

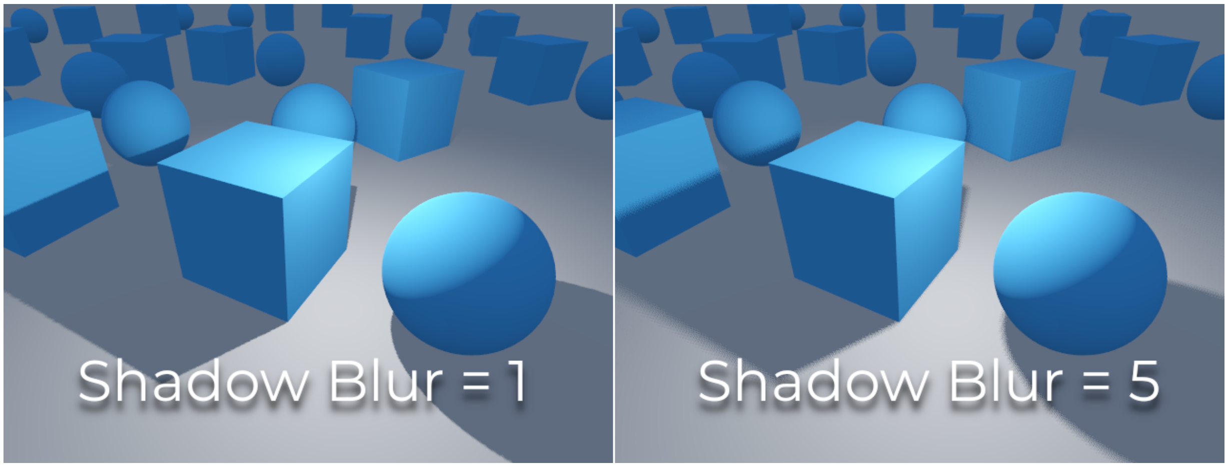
If you need to render more lights than possible in a given rendering backend, consider using baked lightmaps with lights' bake mode set to **Static**. This allows lights to be fully baked, which also makes them much faster to render. You can also use emissive materials with any global illumination technique as a replacement for light nodes that emit light over a large area.

Shadow mapping

Lights can optionally cast shadows. This gives them greater realism (light does not reach occluded areas), but it can incur a bigger performance cost. There is a list of generic shadow parameters, each also has a specific function:

- **Enabled:** Check to enable shadow mapping in this light.
- **Opacity:** Areas occluded are darkened by this opacity factor. Shadows are fully opaque by default, but this can be changed to make shadows translucent for a given light.
- **Bias:** When this parameter is too low, self-shadowing occurs. When too high, shadows separate from the casters. Tweak to what works best for you.
- **Normal Bias:** When this parameter is too low, self-shadowing occurs. When too high, shadows appear misaligned from the casters. Tweak to what works best for you.

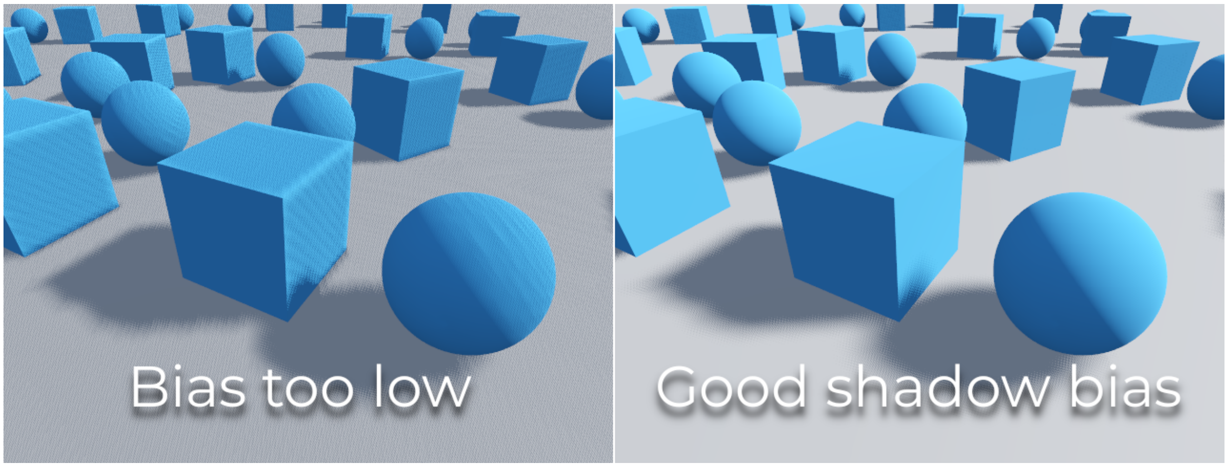
- **Transmittance Bias:** When this parameter is too low, self-shadowing occurs on materials that have transmittance enabled. When too high, shadows will not affect materials that have transmittance enabled consistently. Tweak to what works best for you.
- **Reverse Cull Face:** Some scenes work better when shadow mapping is rendered with face-culling inverted.
- **Blur:** Multiplies the shadow blur radius for this light. This works with both traditional shadow mapping and contact-hardening shadows (lights with **Angular Distance** or **Size** greater than `0.0`). Higher values result in softer shadows, which will also appear to be more temporally stable for moving objects. The downside of increasing shadow blur is that it will make the grainy pattern used for filtering more noticeable. See also Shadow filter mode.



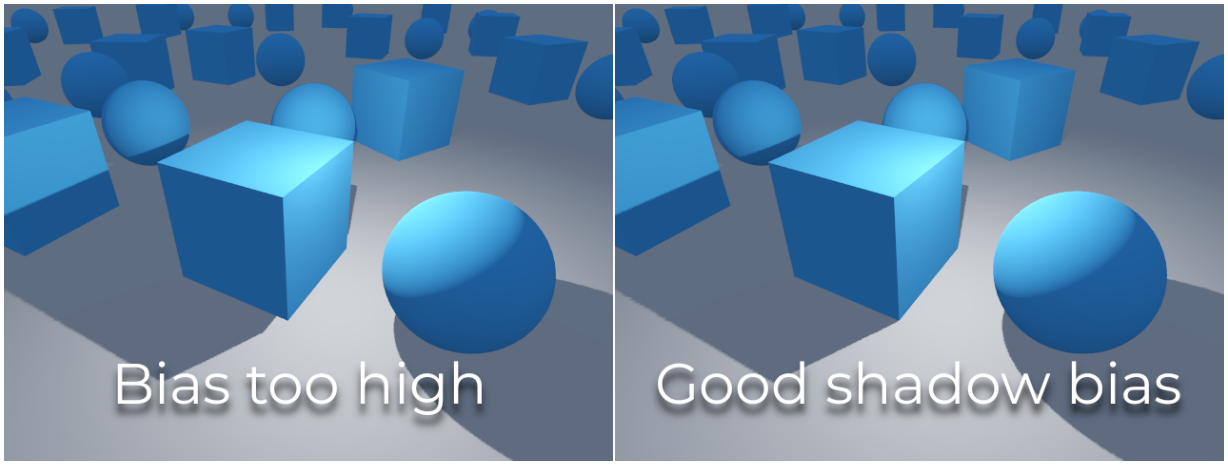
Tweaking shadow bias

Below is an image of what tweaking bias looks like. Default values work for most cases, but in general, it depends on the size and complexity of geometry.

If the **Shadow Bias** or **Shadow Normal Bias** is set too low for a given light, the shadow will be "smeared" onto the objects. This will cause the light's intended appearance to darken, and is called *shadow acne*:



On the other hand, if the **Shadow Bias** or **Shadow Normal Bias** is set too high for a given light, the shadow may appear to be disconnected from the object. This is called *peter-panning*:



In general, increasing **Shadow Normal Bias** is preferred over increasing **Shadow Bias**. Increasing **Shadow Normal Bias** does not cause as much peter-panning as increasing **Shadow Bias**, but it can still resolve most shadow acne issues efficiently. The downside of increasing **Shadow Normal Bias** is that it can make shadows appear thinner for certain objects.

Any sort of bias issues can be fixed by increasing the shadow map resolution, at the cost of decreased performance.

Note

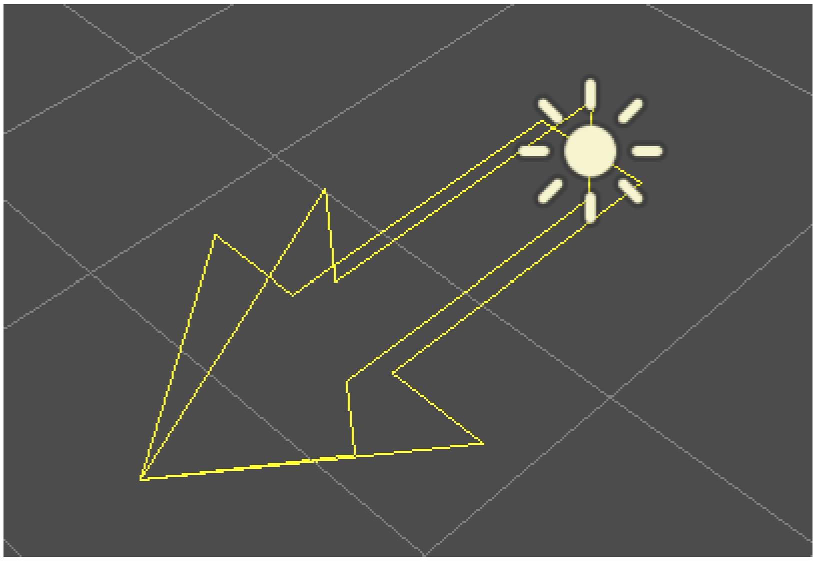
Tweaking shadow mapping settings is an art – there are no "one size fits all" settings. To achieve the best visuals, you may need to use different shadow bias values on a per-light basis.

Note on Appearance Changes: When enabling shadows on a light, be aware that the light's appearance might change compared to when it's rendered without shadows in the compatibility renderer. Due to limitations with older mobile devices, shadows are implemented using a multi-pass rendering approach so lights with shadows are rendered in sRGB space instead of linear space. This change in rendering space can sometimes drastically alter the light's appearance. To achieve a similar appearance to an unshadowed light, you may need to adjust the light's energy setting.

Directional light

This is the most common type of light and represents a light source very far away (such as the sun). It is also the cheapest light to compute and should be used whenever possible (although it's not the cheapest shadow-map to compute, but more on that later).

Directional light models an infinite number of parallel light rays covering the whole scene. The directional light node is represented by a big arrow which indicates the direction of the light rays. However, the position of the node does not affect the lighting at all and can be anywhere.



Every face whose front-side is hit by the light rays is lit, while the others stay dark. Unlike most other light types, directional lights don't have specific parameters.

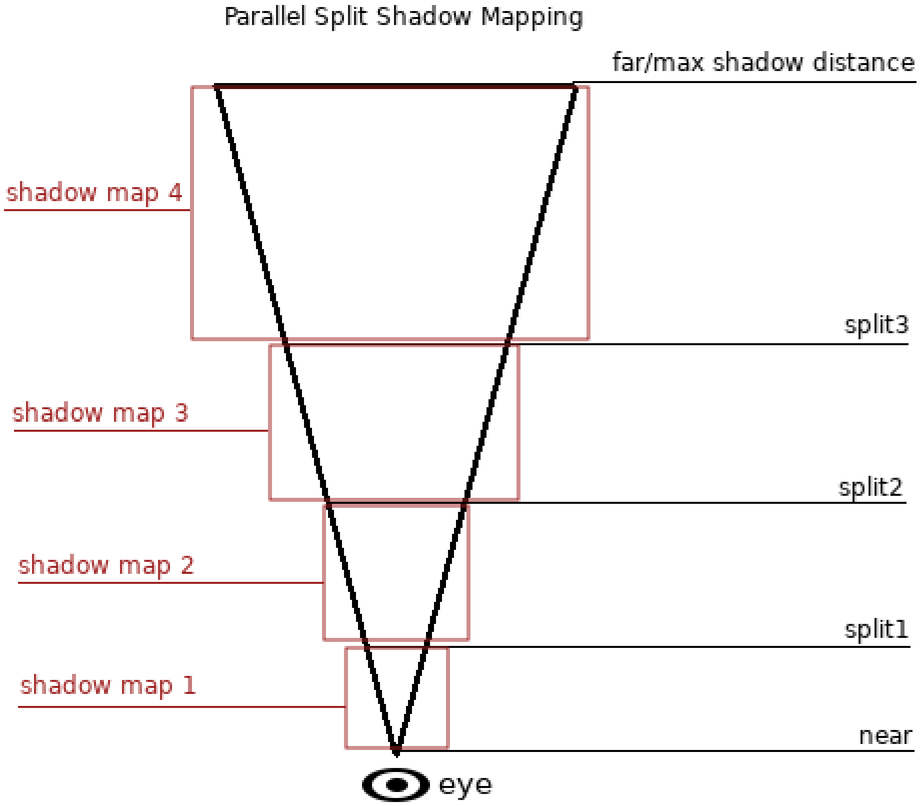
The directional light also offers a **Angular Distance** property, which determines the light's angular size in degrees. Increasing this above `0.0` will make shadows softer at greater distances from the caster, while also affecting the sun's appearance in procedural sky materials. This is called a *contact-hardening* shadow (also known as PCSS).

For reference, the angular distance of the Sun viewed from the Earth is approximately `0.5`. This kind of shadow is expensive, so check the recommendations in PCSS recommendations if setting this value above `0.0` on lights with shadows enabled.

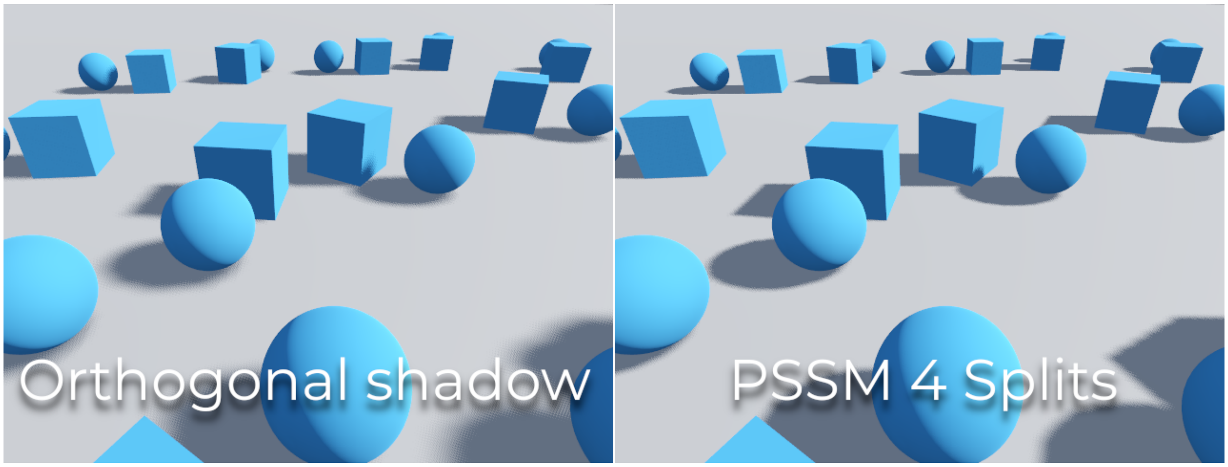
Directional shadow mapping

To compute shadow maps, the scene is rendered (only depth) from an orthogonal point of view that covers the whole scene (or up to the max distance). There is, however, a problem with this approach because objects closer to the camera receive low-resolution shadows that may appear blocky.

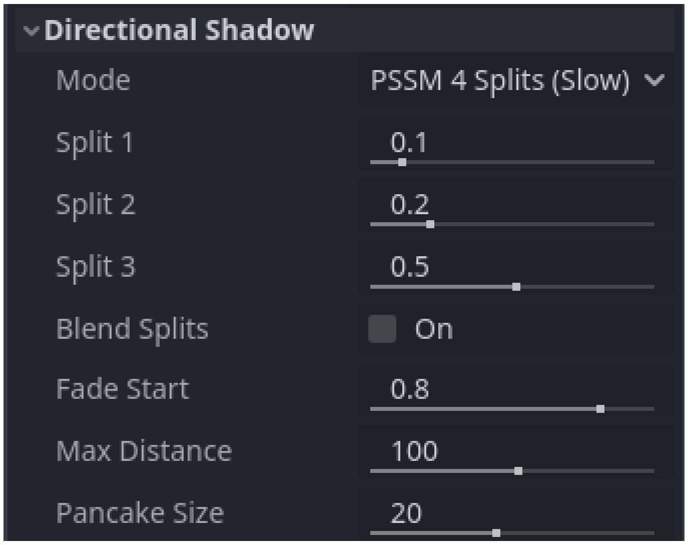
To fix this, a technique named *Parallel Split Shadow Maps* (PSSM) is used. This splits the view frustum in 2 or 4 areas. Each area gets its own shadow map. This allows small areas close to the viewer to have the same shadow resolution as a huge, far-away area. When shadows are enabled for DirectionalLight3D, the default shadow mode is PSSM with 4 splits. In scenarios where an object is large enough to appear in all four splits, it results in increased draw calls. Specifically, such an object will be rendered five times in total: once for each of the four shadow splits and once for the final scene rendering. This can impact performance, understanding this behavior is important for optimizing your scene and managing performance expectations.


Parallel Split Shadow Mapping
far/max shadow distance
shadow map 4
split3
shadow map 3
split2
shadow map 2
split1
shadow map 1
near
eye

With this, shadows become more detailed:



To control PSSM, a number of parameters are exposed:



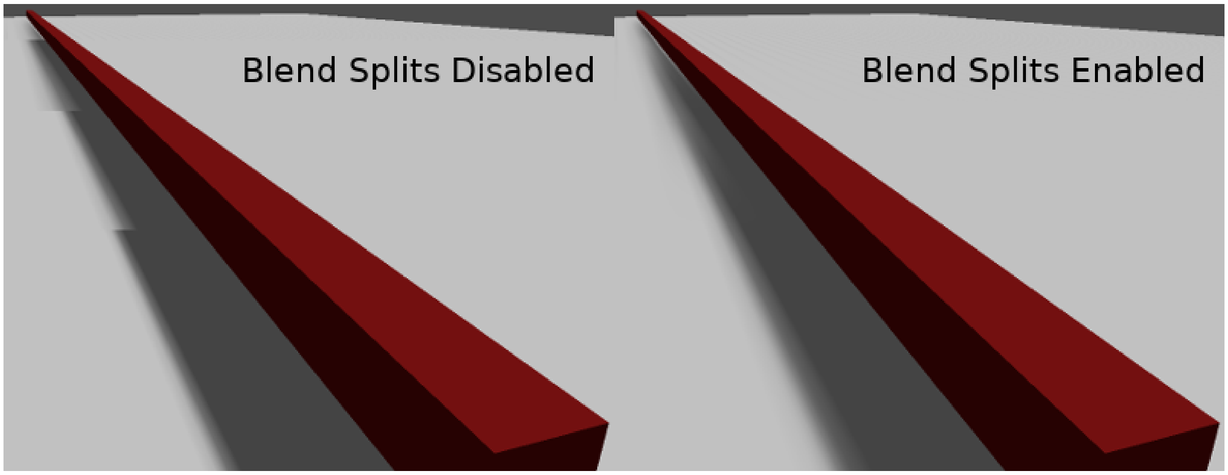
Directional Shadow

Mode	PSSM 4 Splits (Slow)
Split 1	0.1
Split 2	0.2
Split 3	0.5
Blend Splits	☐ On
Fade Start	0.8
Max Distance	100
Pancake Size	20

Each split distance is controlled relative to the camera far (or shadow **Max Distance** if greater than `0.0`). `0.0` is the eye position and `1.0` is where the shadow ends at a distance. Splits are in-between. Default values generally work well, but tweaking the first split a bit is common to give more detail to close objects (like a character in a third-person game).

Always make sure to set a shadow **Max Distance** according to what the scene needs. A lower maximum distance will result in better-looking shadows and better performance, as fewer objects will need to be included in shadow rendering. You can also adjust **Fade Start** to control how aggressive the shadow fade-out should be at a distance. For scenes where the **Max Distance** fully covers the scene at any given camera position, you can increase **Fade Start** to `1.0` to prevent the shadow from fading at a distance. This should not be done in scenes where **Max Distance** doesn't fully cover the scene, as the shadow will appear to be suddenly cut off at a distance.

Sometimes, the transition between a split and the next can look bad. To fix this, the **Blend Splits** option can be turned on, which sacrifices detail and performance in exchange for smoother transitions:

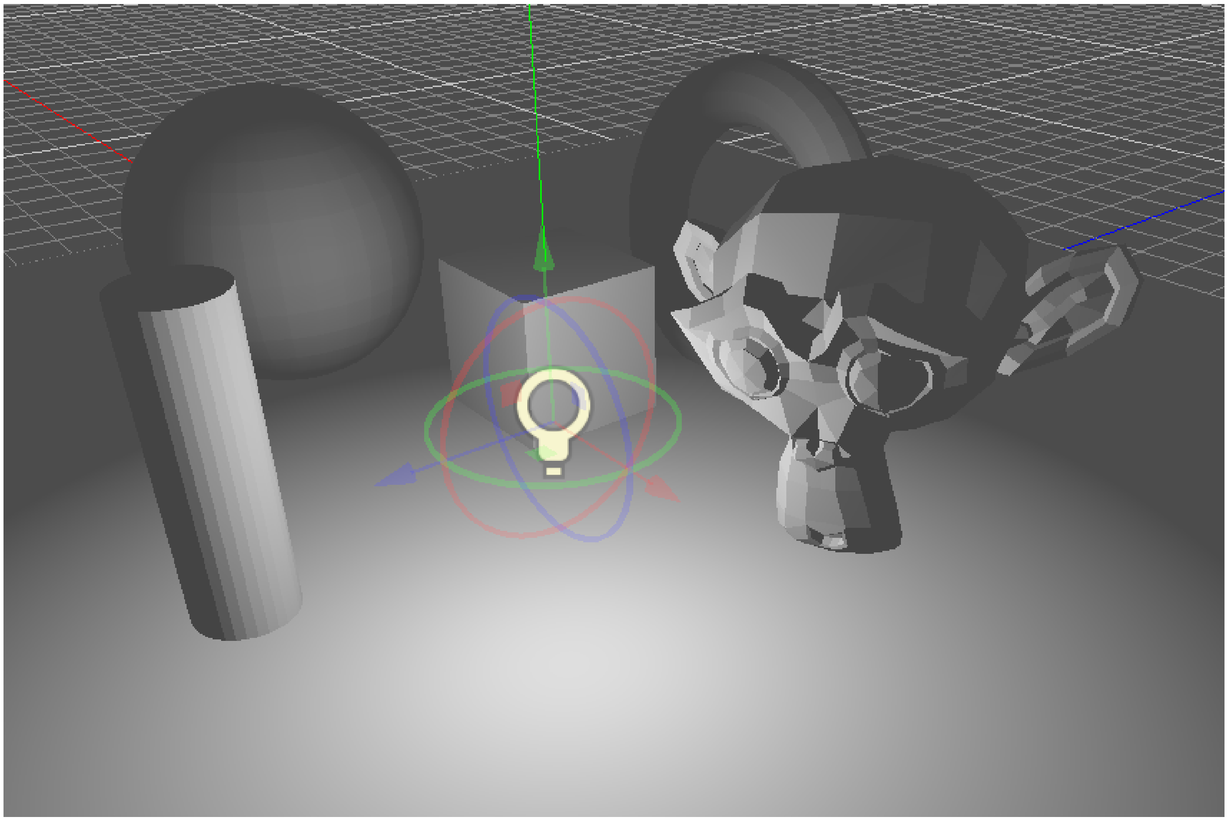


The **Shadow > Normal Bias** parameter can be used to fix special cases of self-shadowing when objects are perpendicular to the light. The only downside is that it makes the shadow a bit thinner. Consider increasing **Shadow > Normal Bias** before increasing **Shadow > Bias** in most situations.

Lastly, **Pancake Size** is a property that can be adjusted to fix missing shadows when using large objects with unsubdivided meshes. Only change this value if you notice missing shadows that are not related to shadow biasing issues.

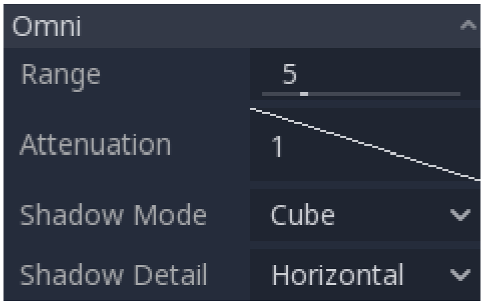
Omni light

Omni light is a point source that emits light spherically in all directions up to a given radius.



In real life, light attenuation is an inverse function, which means omni lights don't have a radius. This is a problem because it means computing several omni lights would become demanding.

To solve this, a **Range** parameter is introduced together with an attenuation function.



Omni

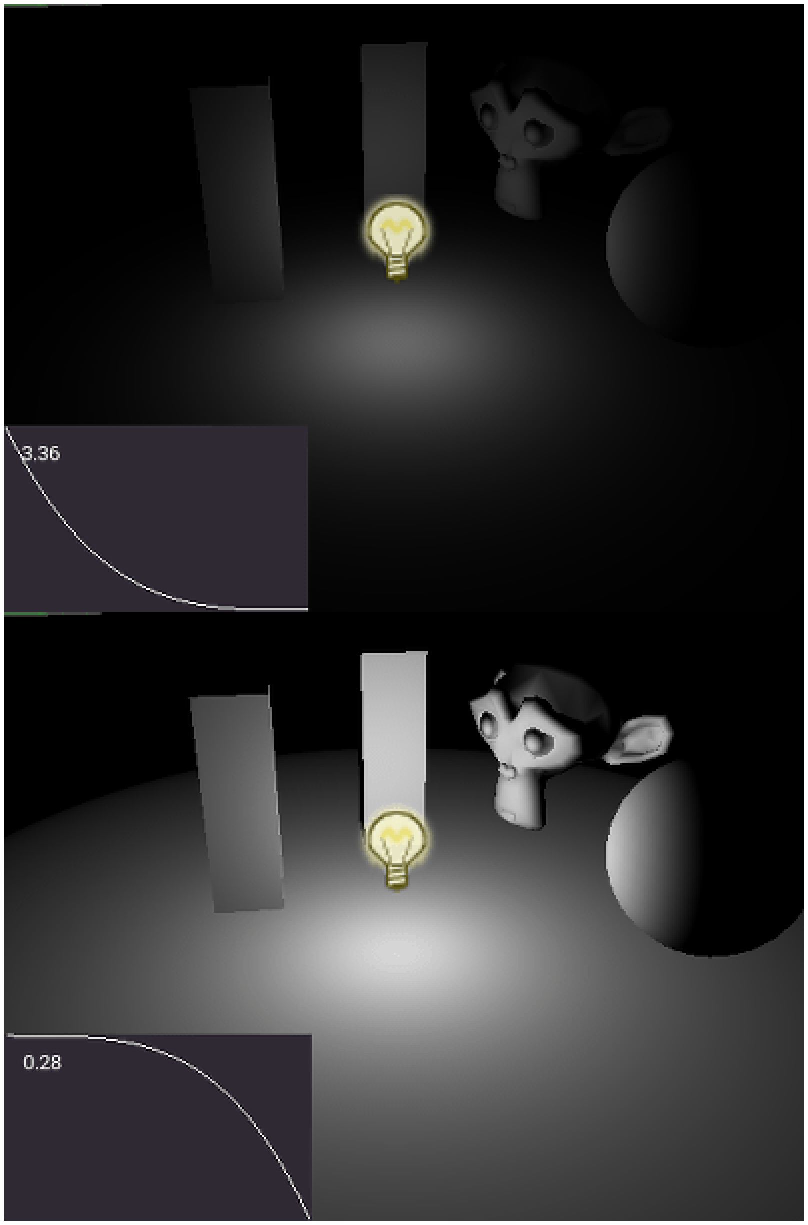
Range 5

Attenuation 1

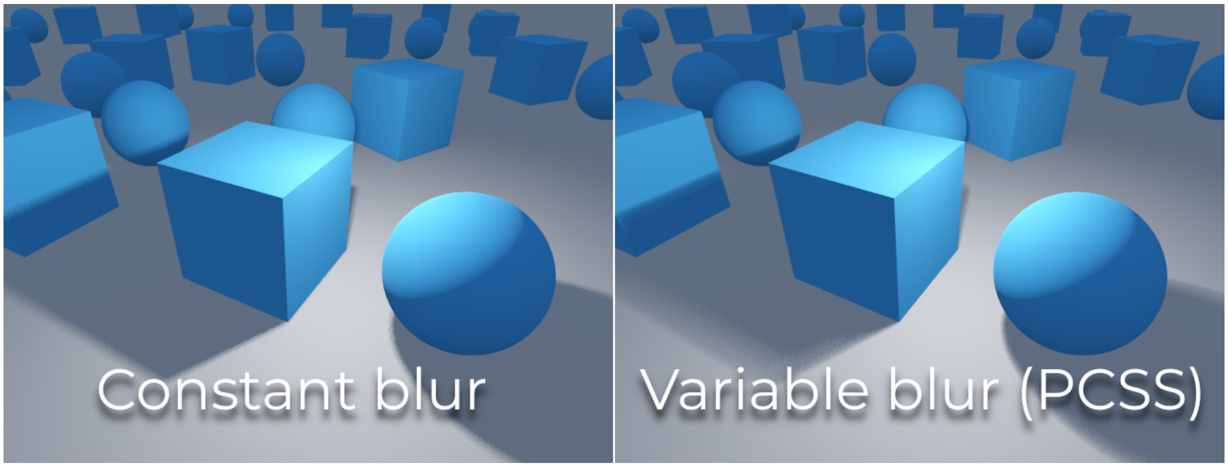
Shadow Mode Cube

Shadow Detail Horizontal

These two parameters allow tweaking how this works visually in order to find aesthetically pleasing results.



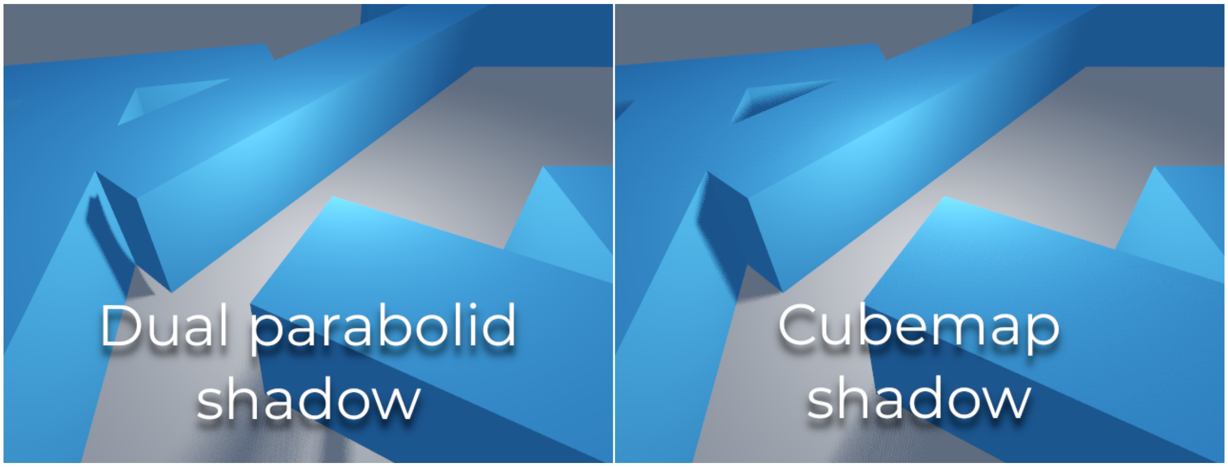
A **Size** parameter is also available in OmniLight3D. Increasing this value will make the light fade out slower and shadows appear blurrier when far away from the caster. This can be used to simulate area lights to an extent. This is called a *contact-hardening* shadow (also known as PCSS). This kind of shadow is expensive, so check the recommendations in PCSS recommendations if setting this value above `0.0` on lights with shadows enabled.



Omni shadow mapping

Omni light shadow mapping is relatively straightforward. The main issue that needs to be considered is the algorithm used to render it.

Omni Shadows can be rendered as either **Dual Paraboloid** or **Cube** mapped. **Dual Parabolid** renders quickly, but can cause deformations, while **Cube** is more correct, but slower. The default is **Cube**, but consider changing it to **Dual Parabolid** for lights where it doesn't make much of a visual difference.

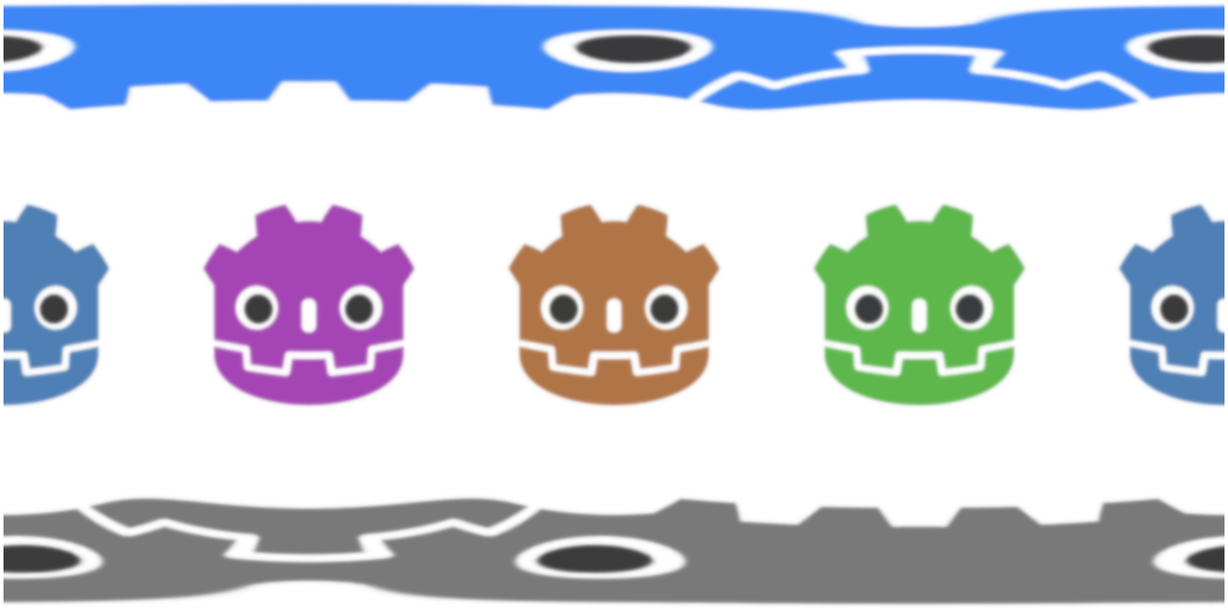


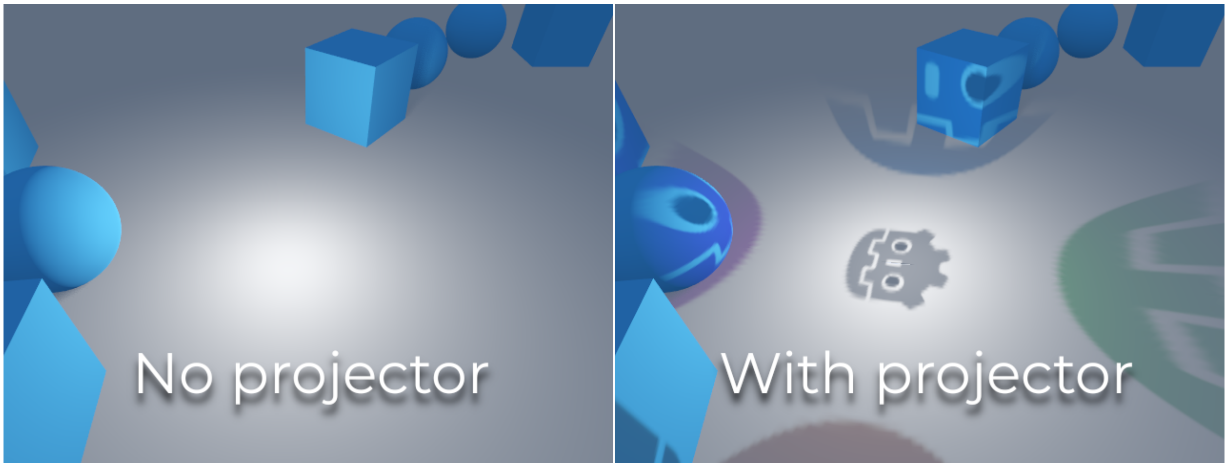
If the objects being rendered are mostly irregular and subdivided, Dual Paraboloid is usually enough. In any case, as these shadows are cached in a shadow atlas (more on that at the end), it may not make a difference in performance for most scenes.

Omni lights with shadows enabled can make use of projectors. The projector texture will *multiply* the light's color by the color at a given point on the texture. As a result, lights will usually appear to be darker once a projector texture is assigned; you can increase **Energy** to compensate for this.

Omni light projector textures require a special 360° panorama mapping, similar to PanoramaSkyMaterial textures.

With the projector texture below, the following result is obtained:





Tip

If you've acquired omni projectors in the form of cubemap images, you can use [this web-based conversion tool](https://danilw.github.io/GLSL-howto/cubemap_to_panorama_js/cubemap_to_panorama.html) [https://danilw.github.io/GLSL-howto/cubemap_to_panorama_js/cubemap_to_panorama.html] to convert them to a single panorama image.

Spot light

Spot lights are similar to omni lights, except they emit light only into a cone (or "cutoff"). They are useful to simulate flashlights, car lights, reflectors, spots, etc. This type of light is also attenuated towards the opposite direction it points to.

Spot lights share the same **Range**, **Attenuation** and **Size** as OmniLight3D, and add two extra parameters:

- **Angle:** The aperture angle of the light.
- **Angle Attenuation:** The cone attenuation, which helps soften the cone borders.

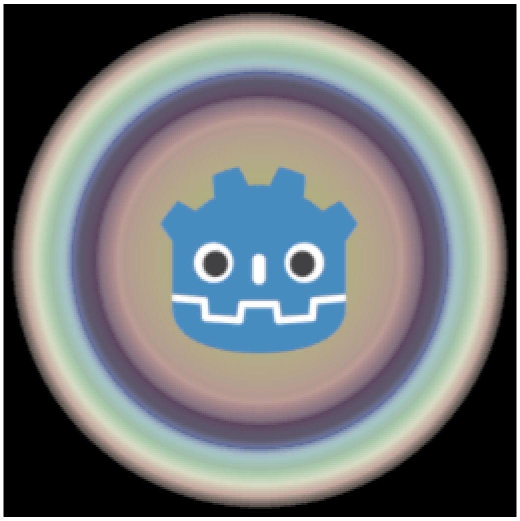
Spot shadow mapping

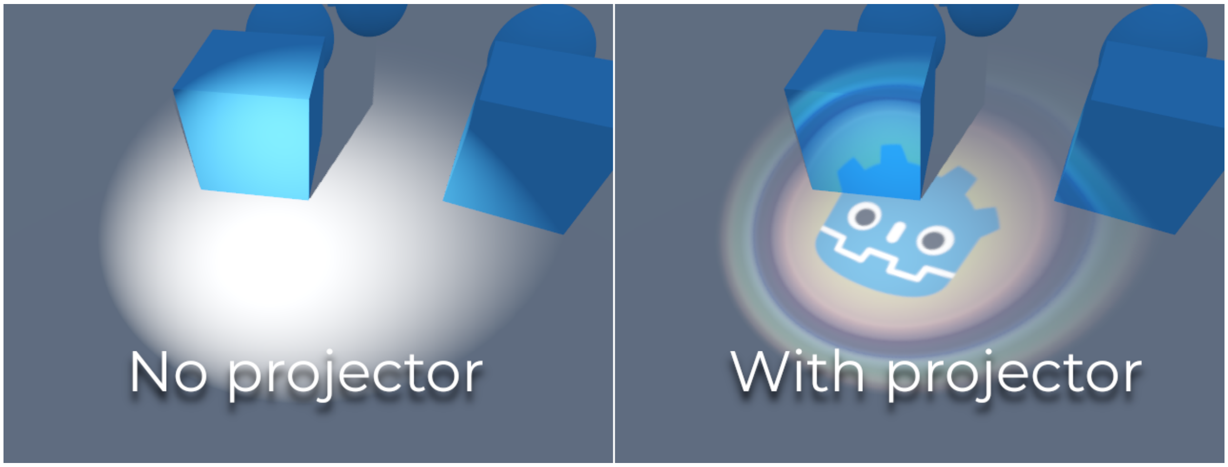
Spots feature the same parameters as omni lights for shadow mapping. Rendering spot shadow maps is significantly faster compared to omni lights, as only one shadow texture needs to be rendered (instead of rendering 6 faces, or 2 in dual parabolid mode).

Spot lights with shadows enabled can make use of projectors. The projector texture will *multiply* the light's color by the color at a given point on the texture. As a result, lights will usually appear to be darker once a projector texture is assigned; you can increase **Energy** to compensate for this.

Unlike omni light projectors, a spot light projector texture doesn't need to follow a special format to look correct. It will be mapped in a way similar to a decal.

With the projector texture below, the following result is obtained:





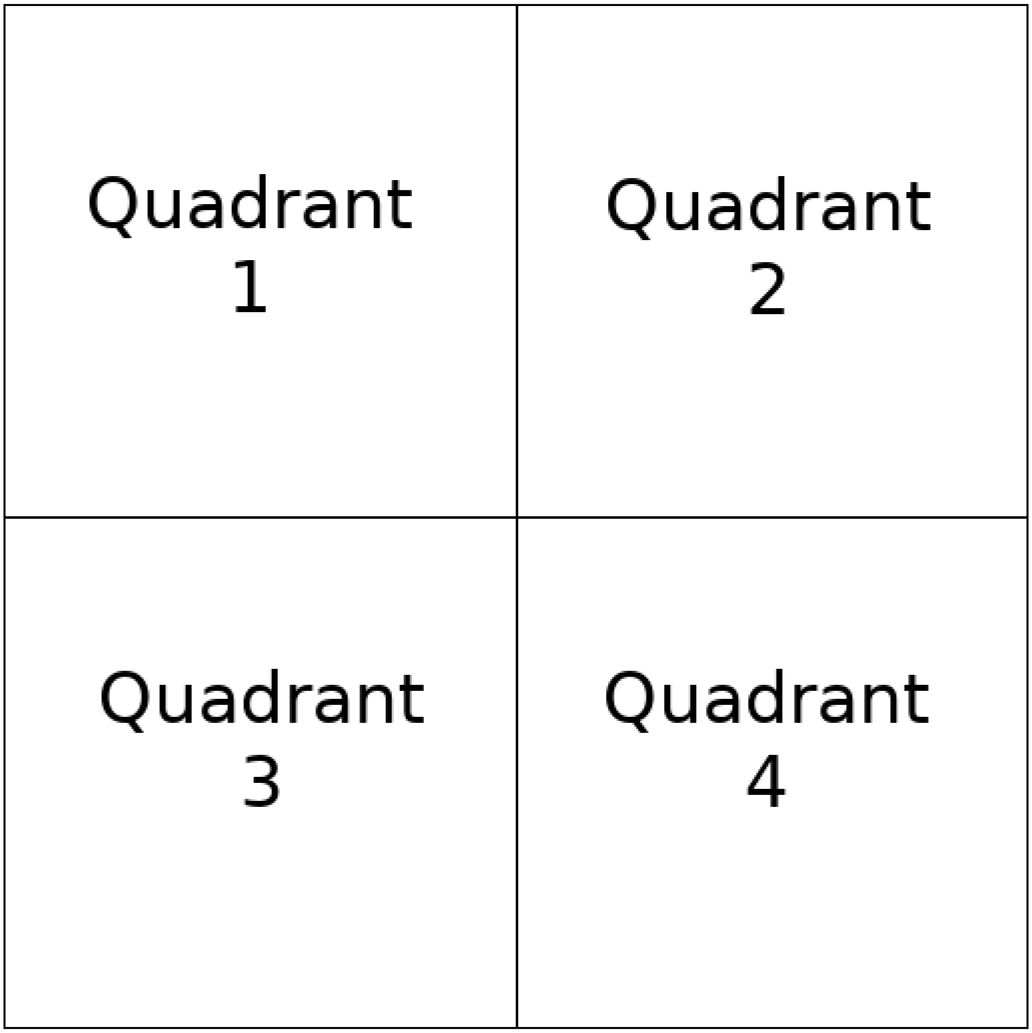
Note

Spot lights with wide angles will have lower-quality shadows than spot lights with narrow angles, as the shadow map is spread over a larger surface. At angles wider than 89 degrees, spot light shadows will stop working entirely. If you need shadows for wider lights, use an omni light instead.

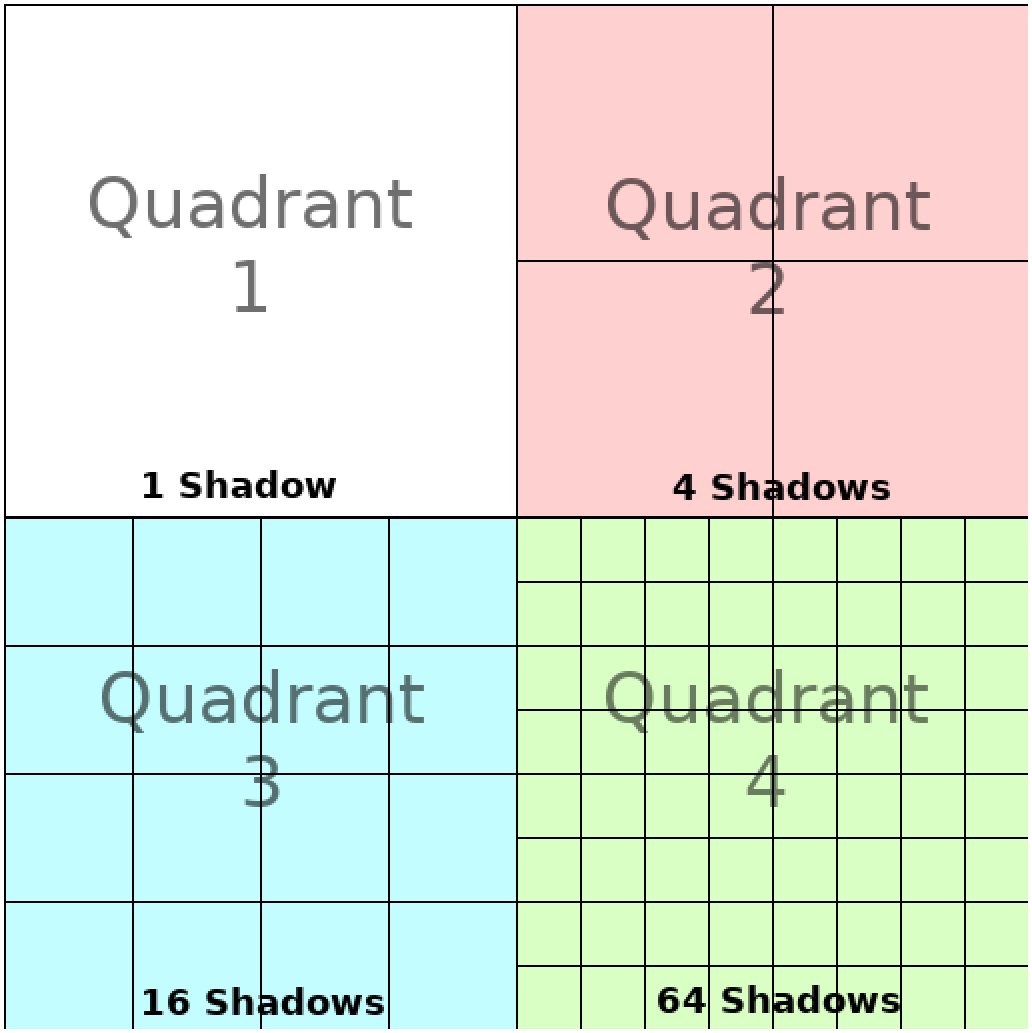
Shadow atlas

Unlike Directional lights, which have their own shadow texture, omni and spot lights are assigned to slots of a shadow atlas. This atlas can be configured in the advanced Project Settings (**Rendering > Lights And Shadows > Positional Shadow**).

The resolution applies to the whole shadow atlas. This atlas is divided into four quadrants:


Quadrant 1
Quadrant 2
Quadrant 3
Quadrant 4

Each quadrant can be subdivided to allocate any number of shadow maps; the following is the default subdivision:


Quadrant 1
1 Shadow
Quadrant 2
4 Shadows
Quadrant 3
16 Shadows
Quadrant 4
64 Shadows

The shadow atlas allocates space as follows:

- The biggest shadow map size (when no subdivision is used) represents a light the size of the screen (or bigger).
- Subdivisions (smaller maps) represent shadows for lights that are further away from view and proportionally smaller.

Every frame, the following procedure is performed for all lights:

1. Check if the light is on a slot of the right size. If not, re-render it and move it to a larger/smaller slot.
2. Check if any object affecting the shadow map has changed. If it did, re-render the light.

3. If neither of the above has happened, nothing is done, and the shadow is left untouched.

If the slots in a quadrant are full, lights are pushed back to smaller slots, depending on size and distance. If all slots in all quadrants are full, some lights will not be able to render shadows even if shadows are enabled on them.

The default shadow allocation strategy allows rendering up to 88 lights with shadows enabled in the camera frustum (4 + 4 + 16 + 64):

1. The first and most detailed quadrant can store 4 shadows.
2. The second quadrant can store 4 other shadows.
3. The third quadrant can store 16 shadows, with less detail.
4. The fourth and least detailed quadrant can store 64 shadows, with even less detail.

Using a higher number of shadows per quadrant allows supporting a greater amount of total lights with shadows enabled, while also improving performance (as shadows will be rendered at a lower resolution for each light). However, increasing the number of shadows per quadrant comes at the cost of lower shadow quality.

In some cases, you may want to use a different allocation strategy. For example, in a top-down game where all lights are around the same size, you may want to set all quadrants to have the same subdivision so that all lights have shadows of similar quality level.

Balancing performance and quality

Shadow rendering is a critical topic in 3D rendering performance. It's important to make the right choices here to avoid creating bottlenecks.

Directional shadow quality settings can be changed at run-time by calling the appropriate RenderingServer methods.

Positional (omni/spot) shadow quality settings can be changed at run-time on the root Viewport.

Shadow map size

High shadow resolutions result in sharper shadows, but at a significant performance cost. It should also be noted that *sharper shadows are not always more realistic*. In most cases, this should be kept at its default value of `4096` or decreased to `2048` for low-end GPUs.

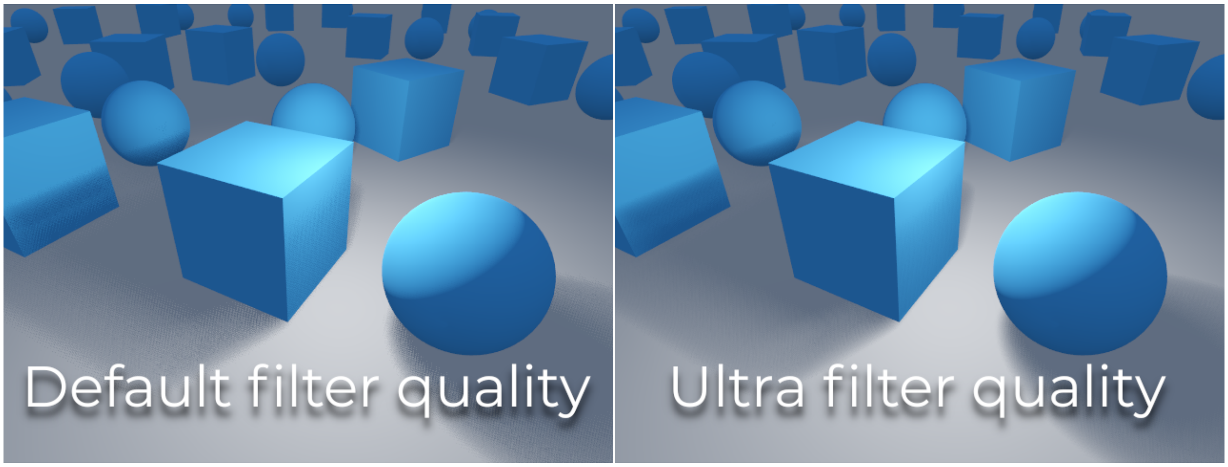
If positional shadows become too blurry after decreasing the shadow map size, you can counteract this by adjusting the shadow atlas quadrants to contain fewer shadows. This will allow each shadow to be rendered at a higher resolution.

Shadow filter mode

Several shadow map quality settings can be chosen here. The default **Soft Low** is a good balance between performance and quality for scenes with detailed textures, as the texture detail will help make the dithering pattern less noticeable.

However, in projects with less detailed textures, the shadow dithering pattern may be more visible. To hide this pattern, you can either enable Temporal antialiasing (TAA), AMD FidelityFX Super Resolution 2.2 (FSR2), Fast approximate antialiasing (FXAA), or increase the shadow filter quality to **Soft Medium** or higher.

The **Soft Very Low** setting will automatically decrease shadow blur to make artifacts from the low sample count less visible. Conversely, the **Soft High** and **Soft Ultra** settings will automatically increase shadow blur to better make use of the increased sample count.



16-bits versus 32-bit

By default, Godot uses 16-bit depth textures for shadow map rendering. This is recommended in most cases as it performs better without a noticeable difference in quality.

If **16 Bits** is disabled, 32-bit depth textures will be used instead. This can result in less artifacting in large scenes and large lights with shadows enabled. However, the difference is often barely visible, yet this can have a significant performance cost.

Light/shadow distance fade

OmniLight3D and SpotLight3D offer several properties to hide distant lights. This can improve performance significantly in large scenes with dozens of lights or more.

- **Enabled:** Controls whether distance fade (a form of LOD) is enabled. The light will fade out over **Begin + Length**, after which it will be culled and not sent to the shader at all. Use this to reduce the number of active lights in a scene and thus improve performance.

- **Begin:** The distance from the camera at which the light begins to fade away (in 3D units).
- **Shadow:** The distance from the camera at which the shadow begins to fade away (in 3D units). This can be used to fade out shadows sooner compared to the light, further improving performance. Only available if shadows are enabled for the light.
- **Length:** The distance over which the light and shadow fades (in 3D units). The light becomes slowly more transparent over this distance and is completely invisible at the end. Higher values result in a smoother fade-out transition, which is more suited when the camera moves fast.

PCSS recommendations

Percentage-closer soft shadows (PCSS) provide a more realistic shadow mapping appearance, with the penumbra size varying depending on the distance between the caster and the surface receiving the shadow. This comes at a high performance cost, especially for directional lights.

To avoid performance issues, it's recommended to:

- Only use a handful of lights with PCSS shadows enabled at a given time. The effect is generally most visible on large, bright lights. Secondary light sources that are more faint usually don't benefit much from using PCSS shadows.
- Provide a setting for users to disable PCSS shadows. On directional lights, this can be done by setting the DirectionalLight3D's `light_angular_distance` property to `0.0` in a script. On positional lights, this can be done by setting the OmniLight3D or SpotLight3D's `light_size` property to `0.0` in a script.

Projector filter mode

The way projectors are rendered also has an impact on performance. The **Rendering > Textures > Light Projectors > Filter** advanced project setting lets you control how projector textures should be filtered. **Nearest/Linear** do not use mipmaps, which makes them faster to render. However, projectors will look grainy at distance. **Nearest/Linear Mipmaps** will look smoother at a distance, but projectors will look blurry when viewed from oblique angles. This can be resolved by using **Nearest/Linear Mipmaps Anisotropic**, which is the highest-quality mode but also the most expensive.

If your project has a pixel art style, consider setting the filter to one of the **Nearest** values so that projectors use nearest-neighbor filtering. Otherwise, stick to **Linear**.

Using decals

Note

Decals are only supported in the Clustered Forward and Forward Mobile rendering backends, not the Compatibility backend.

If using the Compatibility backend, consider using Sprite3D as an alternative for projecting decals onto (mostly) flat surfaces.

Decals are projected textures that apply on opaque or transparent surfaces in 3D. This projection happens in real-time and doesn't rely on mesh generation. This allows you to move decals every frame with only a small performance impact, even when applied on complex meshes.

While decals cannot add actual geometry detail onto the projected surface, decals can still make use of physically-based rendering to provide similar properties to full-blown PBR materials.

On this page, you'll learn:

- How to set up decals in the 3D editor.
- How to create decals during gameplay in a 3D scene (such as bullet impacts).
- How to balance decal configuration between performance and quality.

See also

The Godot demo projects repository contains a 3D decals demo [https://github.com/godotengine/godot-demo-projects/tree/master/3d/decals].

If you're looking to write arbitrary 3D text on top of a surface, use 3D text placed close to a surface instead of a Decal node.

Use cases

Static decoration

Sometimes, the fastest way to add texture detail to a scene is to use decals. This is especially the case for organic detail, such as patches of dirt or sand scattered on a large surface. Decals can help break up texture repetition in scenes and make patterns look more natural. On a smaller scale, decals can also be used to create detail variations for objects. For example, decals can be used to add nuts and bolts on top of hard-surface geometry.

Since decals can inject their own PBR properties on top of the projected surfaces, they can also be used to create footprints or wet puddles.

Dirt added on top of level geometry using decals

Dynamic gameplay elements

Decals can represent temporary or persistent gameplay effects such as bullet impacts and explosion scorches.

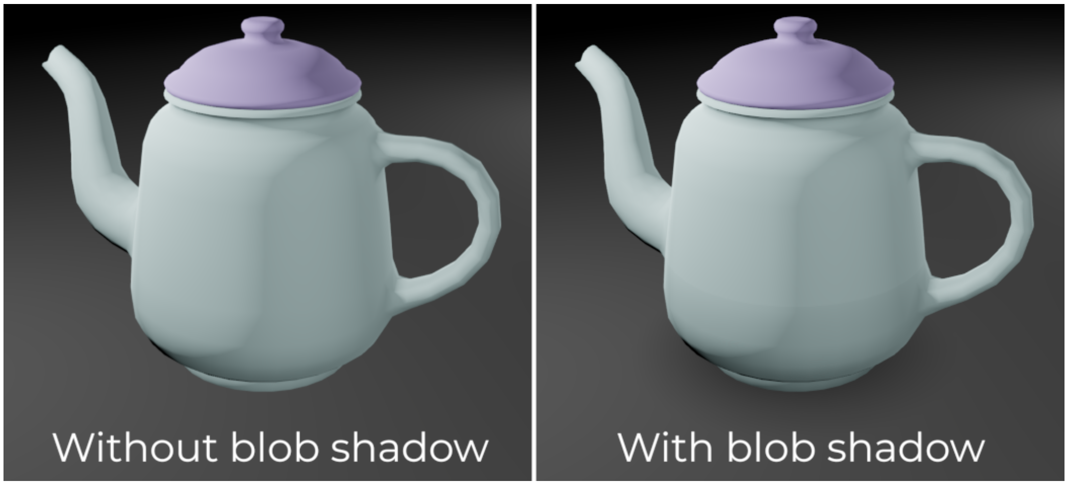
Using an AnimationPlayer node or a script, decals can be made to fade over time (and then be removed using `queue_free()`) to improve performance.

Blob shadows

Blob shadows are frequently used in mobile projects (or to follow a retro art style), as real-time lighting tends to be too expensive on low-end mobile devices. However, when relying on baked lightmaps with fully baked lights, dynamic objects will not cast *any* shadow from those lights. This makes dynamic objects in lightmapped scenes look flat in comparison to real-time lighting, with dynamic objects almost looking like they're floating.

Thanks to blob shadows, dynamic objects can still cast an approximative shadow. Not only this helps with depth perception in the scene, but this can also be a gameplay element, especially in 3D platformers. The blob shadow's length can be extended to let the player know where they will land if they fall straight down.

Even with real-time lighting, blob shadows can still be useful as a form of ambient occlusion for situations where SSAO is too expensive or too unstable due to its screen-space nature. For example, vehicles' underside shadows are well-represented using blob shadows.



Blob shadow under object comparison

Quick start guide

Creating decals in the editor

1. Create a Decal node in the 3D editor.
2. In the inspector, expand the **Textures** section and load a texture in **Textures > Albedo**.
3. Move the Decal node towards an object, then rotate it so the decal is visible (and in the right orientation). If the decal appears mirrored, try to rotate it by 180 degrees. You can double-check whether it's in the right orientation by increasing **Parameters > Normal Fade** to 0.5. This will prevent the Decal from being projected on surfaces that are not facing the decal.
4. If your decal is meant to affect static objects only, configure it to prevent affecting dynamic objects (or vice versa). To do so, change the decal's **Cull Mask** property to exclude certain layers. After doing this, modify your dynamic objects' MeshInstance3D nodes to change their visibility layers. For instance, you can move them from layer 1 to layer 2, then disable layer 2 in the decal's **Cull Mask** property.

Decal node properties

- **Extents:** The size of the decal. The Y axis determines the length of the decal's projection. Keep the projection length as short as possible to improve culling opportunities, therefore improving performance.

Textures

- **Albedo:** The albedo (diffuse/color) map to use for the decal. In most situations, this is the texture you want to set first. If using a normal or ORM map, an albedo map *must* be set to provide an alpha channel. This alpha channel will be used as a mask to determine how much the normal/ORM maps will affect the underlying surface.

- **Normal:** The normal map to use for the decal. This can be used to increase perceived detail on the decal by modifying how light reacts to it. The impact of this texture is multiplied by the albedo texture's alpha channel (but not **Albedo Mix**).
- **ORM:** The Occlusion/Roughness/Metallic map to use for the decal. This is an optimized format for storing PBR material maps. Ambient Occlusion map is stored in the red channel, roughness map in the green channel, metallic map in the blue channel. The impact of this texture is multiplied by the albedo texture's alpha channel (but not **Albedo Mix**).
- **Emission:** The emission texture to use for the decal. Unlike **Albedo**, this texture will appear to glow in the dark.

Parameters

- **Emission Energy:** The brightness of the emission texture.
- **Modulate:** Multiplies the color of the albedo and emission textures. Use this to tint decals (e.g. for paint decals, or to increase variation by randomizing each decal's modulation).
- **Albedo Mix:** The opacity of the albedo texture. Unlike using an albedo texture with a more transparent alpha channel, decreasing this value below `1.0` does *not* reduce the impact of the normal/ORM texture on the underlying surface. Set this to `0.0` when creating normal/ORM-only decals such as footsteps or wet puddles.
- **Normal Fade:** Fades the Decal if the angle between the Decal's AABB and the target surface becomes too large. A value of `0.0` projects the decal regardless of angle, while a value of `0.999` limits the decal to surfaces that are nearly perpendicular. Setting **Normal Fade** to a value greater than `0.0` has a small performance cost due to the added normal angle computations.

Vertical Fade

- **Upper Fade:** The curve over which the decal will fade as the surface gets further from the center of the AABB (towards the decal's projection angle). Only positive values are valid.
- **Lower Fade:** The curve over which the decal will fade as the surface gets further from the center of the AABB (away from the decal's projection angle). Only positive values are valid.

Distance Fade

- **Enabled:** Controls whether distance fade (a form of LOD) is enabled. The decal will fade out over **Begin + Length**, after which it will be culled and not sent to the shader at all. Use this to reduce the number of active decals in a scene and thus improve performance.
- **Begin:** The distance from the camera at which the decal begins to fade away (in 3D units).
- **Length:** The distance over which the decal fades (in 3D units). The decal becomes slowly more transparent over this distance and is completely invisible at the end. Higher values result in a smoother fade-out transition, which is more suited when the camera moves fast.

Cull Mask

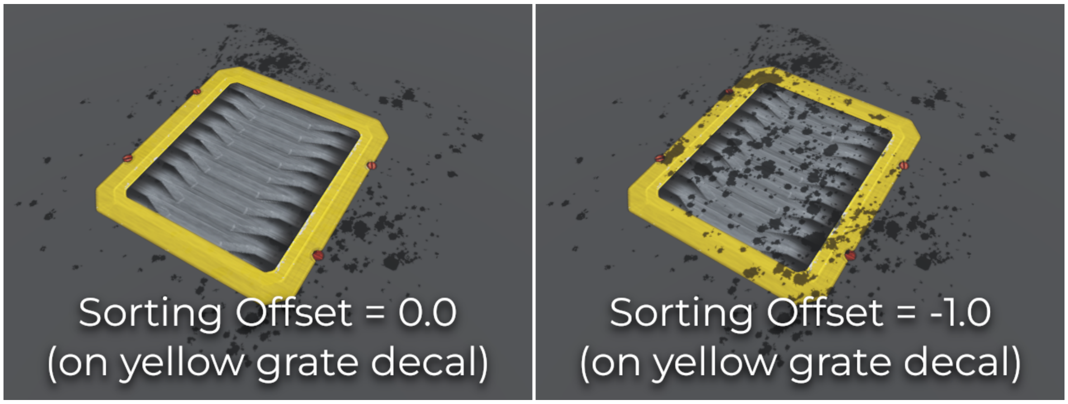
- **Cull Mask:** Specifies which VisualInstance3D layers this decal will project on. By default, decals affect all layers. This is used so you can specify which types of objects receive the decal and which do not. This is especially useful so you can ensure that dynamic objects don't accidentally receive a Decal intended for the terrain under them.

Decal rendering order

By default, decals are ordered based on the size of their AABB and the distance to the camera. AABBs that are closer to the camera are rendered first, which means that decal rendering order can sometimes appear to change depending on camera position if some decals are positioned at the same location.

To resolve this, you can adjust the **Sorting Offset** property in the VisualInstance3D section of the Decal node inspector. This offset is not a strict priority order, but a *guideline* that the renderer will use as the AABB size still affects how decal sorting works. Therefore, higher values will *always* result in the decal being drawn above other decals with a lower sorting offset.

If you want to ensure a decal is always rendered on top of other decals, you need to set its **Sorting Offset** property to a positive value greater than the AABB length of the largest decal that may overlap it. To make this decal drawn behind other decals instead, set the **Sorting Offset** to the same negative value.



VisualInstance3D Sorting Offset comparison on Decals

Tweaking performance and quality

Decal rendering performance is mostly determined by their screen coverage, but also their number. In general, a few large decals that cover up most of the screen will be more expensive to render than many small decals that are scattered around.

To improve rendering performance, you can enable the **Distance Fade** property as described above. This will make distant decals fade out when they are far away from the camera (and may have little to no impact on the final scene rendering). Using node groups, you can also prevent non-essential decorative decals from spawning based on user configuration.

The way decals are rendered also has an impact on performance. The **Rendering > Textures > Decals > Filter** advanced project setting lets you control how decal textures should be filtered. **Nearest/Linear** does not use mipmaps. However, decals will look grainy at a distance. **Nearest/Linear Mipmaps** will look smoother at a distance, but decals will look blurry when viewed from oblique angles. This can be resolved by using **Nearest/Linear Mipmaps Anisotropic**, which provides the highest quality but is also slower to render.

If your project has a pixel art style, consider setting the filter to one of the **Nearest** values so that decals use nearest-neighbor filtering. Otherwise, stick to **Linear**.

Limitations

Decals cannot affect material properties other than the ones listed above, such as height (for parallax mapping).

For performance reasons, decals use purely fixed rendering logic. This means decals cannot use custom shaders. However, custom shaders on the projected surfaces are able to read the information that is overridden by decals on top of them, such as roughness and metallic.

When using the Forward+ renderer, Godot uses a *clustering* approach for decal rendering. As many decals as desired can be added (as long as performance allows). However, there's still a default limit of 512 *clustered elements* that can be present in the current camera view. A clustered element is an omni light, a spot light, a decal or a reflection probe. This limit can be increased by adjusting the **Rendering > Limits > Cluster Builder > Max Clustered Elements** advanced project setting.

When using the Forward Mobile backend, only 8 decals can be applied on each individual Mesh *resource*. If there are more decals affecting a single mesh, not all of them will be rendered on the mesh.

Physical light and camera units

Why use physical light and camera units?

Godot uses arbitrary units for many physical properties that apply to light like color, energy, camera field of view, and exposure. By default, these properties use arbitrary units, because using accurate physical units comes with a few tradeoffs that aren't worth it for many games. As Godot favors ease of use by default, physical light units are disabled by default.

Advantages of physical units

If you aim for photorealism in your project, using real world units as a basis can help make things easier to adjust. References for real world materials, lights and scene brightness are wildly available on websites such as Physically Based [https://physicallybased.info/].

Using real world units in Godot can also be useful when porting a scene from other 3D software that uses physical light units (such as Blender).

Disadvantages of physical units

The biggest disadvantage of using physical light units is you will have to pay close attention to the dynamic range in use at a given time. You can run into floating point precision errors when mixing very high light intensities with very low light intensities.

In practice, this means that you will have to manually manage your exposure settings to ensure that you aren't over-exposing or under-exposing your scene too much. Auto-exposure can help you balance the light in a scene to bring it into a normal range, but it can't recover lost precision from a dynamic range that is too high.

Using physical light and camera units will not automatically make your project look *better*. Sometimes, moving away from realism can actually make a scene look better to the human eye. Also, using physical units requires a greater amount of rigor compared to non-physical units. Most benefits of physical units can only be obtained if the units are correctly set to match real world reference.

Note

Physical light units are only available in 3D rendering, not 2D.

Setting up physical light units

Physical light units can be enabled separately from physical camera units.

To enable physical light units correctly, there are 4 steps required:

1. Enable the project setting.
2. Configure the camera.
3. Configure the environment.
4. Configure Light3D nodes.

Since physical light and camera units only require a handful of calculations to handle unit conversion, enabling them doesn't have any noticeable performance impact on the CPU. However, on the GPU side, physical camera units currently enforce depth of field. This has a moderate performance impact. To alleviate this performance impact, depth of field quality can be decreased in the advanced Project Settings.

Enable the project setting

Open the Project Settings, enable the **Advanced** toggle then enable **Rendering > Lights And Shadows > Use Physical Light Units**. Restart the editor.

Configure the camera

Warning

When physical light units are enabled and if you have a WorldEnvironment node in your scene (i.e. the editor Environment is disabled), you **must** have a CameraAttributes resource assigned to the WorldEnvironment node. Otherwise, the 3D editor viewport will appear extremely bright if you have a visible DirectionalLight3D node.

On the Camera3D node, you can add a CameraAttributes resource to its **Attributes** property. This resource is used to control the camera's depth of field and exposure. When using CameraAttributesPhysical, its focal length property is also used to adjust the camera's field of view.

When physical light units are enabled, the following additional properties become available in CameraAttributesPhysical's **Exposure** section:

- **Aperture:** The size of the aperture of the camera, measured in f-stops. An f-stop is a unitless ratio between the focal length of the camera and the diameter of the aperture. A high aperture setting will result in a smaller aperture which leads to a dimmer image and sharper focus. A low aperture results in a wide aperture which lets in more light resulting in a brighter, less-focused image.

- **Shutter Speed:** The time for shutter to open and close, measured in *inverse seconds* (`1/N`). A lower value will let in more light leading to a brighter image, while a higher value will let in less light leading to a darker image. *When getting or setting this property with a script, the unit is in seconds instead of inverse seconds.*
- **Sensitivity:** The sensitivity of camera sensors, measured in ISO. A higher sensitivity results in a brighter image. When auto exposure is enabled, this can be used as a method of exposure compensation. Doubling the value will increase the exposure value (measured in EV100) by 1 stop.
- **Multiplier:** A *non-physical* exposure multiplier. Higher values will increase the scene's brightness. This can be used for post-processing adjustments or for animation purposes.

The default **Aperture** value of 16 f-stops is appropriate for outdoors at daytime (i.e. for use with a default DirectionalLight3D). For indoor lighting, a value between 2 and 4 is more appropriate.

Typical shutter speed used in photography and movie production is 1/50 (0.02 seconds). Night-time photography generally uses a shutter around 1/10 (0.1 seconds), while sports photography uses a shutter speed between 1/250 (0.004 seconds) and 1/1000 (0.001 seconds) to reduce motion blur.

In real life, sensitivity is usually set between 50 ISO and 400 ISO for daytime outdoor photography depending on weather conditions. Higher values are used for indoor or night-time photography.

Note

Unlike real life cameras, the adverse effects of increasing ISO sensitivity or decreasing shutter speed (such as visible grain or light trails) are not simulated in Godot.

See Setting up physical camera units for a description of CameraAttributesPhysical properties that are also available when **not** using physical light units.

Configure the environment

Warning

The default configuration is designed for daytime outdoor scenes. Night-time and indoor scenes will need adjustments to the DirectionalLight3D and WorldEnvironment background intensity to look correct. Otherwise, positional lights will be barely visible at their default intensity.

If you haven't added a WorldEnvironment and Camera3D node to the current scene yet, do so now by clicking the 3 vertical dots at the top of the 3D editor viewport. Click **Add Sun to Scene**, open the dialog again then click **Add Environment to Scene**.

After enabling physical light units, a new property becomes available to edit in the Environment resource:

- **Background Intensity:** The background sky's intensity in nits [https://en.wikipedia.org/wiki/Candela_per_square_metre] (candelas per square meter). This also affects ambient and reflected light if their respective modes are set to **Background**. If a custom **Background Energy** is set, this energy is multiplied by the intensity.

Configure the light nodes

After enabling physical light units, 2 new properties become available in Light3D nodes:

- **Intensity:** The light's intensity in lux [https://en.wikipedia.org/wiki/Lux] (DirectionalLight3D) or lumens [https://en.wikipedia.org/wiki/Lumen_(unit)] (OmniLight3D/SpotLight3D). If a custom **Energy** is set, this energy is multiplied by the intensity.
- **Temperature:** The light's *color temperature* defined in Kelvin. If a custom **Color** is set, this color is multiplied by the color temperature.

OmniLight3D/SpotLight3D intensity

Lumens are a measure of luminous flux, which is the total amount of visible light emitted by a light source per unit of time.

For SpotLight3Ds, we assume that the area outside the visible cone is surrounded by a perfect light absorbing material. Accordingly, the apparent brightness of the cone area does *not* change as the cone increases and decreases in size.

A typical household lightbulb can range from around 600 lumens to 1200 lumens. A candle is about 13 lumens, while a streetlight can be approximately 60000 lumens.

DirectionalLight3D intensity

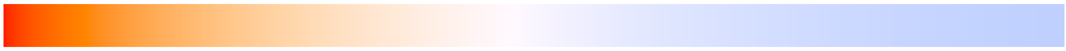
Lux is a measure pf luminous flux per unit area, it is equal to one lumen per square metre. Lux is the measure of how much light hits a surface at a given time.

With DirectionalLight3D, on a clear sunny day, a surface in direct sunlight may receive approximately 100000 lux. A typical room in a home may receive approximately 50 lux, while the moonlit ground may receive approximately 0.1 lux.

Color temperature

6500 Kelvin is white. Higher values result in colder (bluer) colors, while lower values result in warmer (more orange) colors.

The sun on a cloudy day is approximately 6500 Kelvin. On a clear day, the sun is between 5500 to 6000 Kelvin. On a clear day at sunrise or sunset, the sun ranges to around 1850 Kelvin.



Color temperature chart from 1,000 Kelvin (left) to 12,500 Kelvin (right)

Other Light3D properties such as **Energy** and **Color** remain editable for animation purposes, and when you occasionally need to create lights with non-realistic properties.

Setting up physical camera units

Physical camera units can be enabled separately from physical light units.

After adding a CameraAttributesPhysical resource to the **Camera Attributes** property of a Camera3D nodes, some properties such as **FOV** will no longer be editable. Instead, these properties are now governed by the CameraAttributesPhysical's properties, such as focal length and aperture.

CameraAttributesPhysical offers the following properties in its **Frustum** section:

- **Focus Distance:** Distance from camera of object that will be in focus, measured in meters. Internally, this will be clamped to be at least 1 millimeter larger than the **Focal Length**.

- **Focal Length:** Distance between camera lens and camera aperture, measured in millimeters. Controls field of view and depth of field. A larger focal length will result in a smaller field of view and a narrower depth of field meaning fewer objects will be in focus. A smaller focal length will result in a wider field of view and a larger depth of field, which means more objects will be in focus. This property overrides the Camera3D's **FOV** and **Keep Aspect** properties, making them read-only in the inspector.
- **Near/Far:** The near and far clip distances in meters. These behave the same as the Camera3D properties of the same name. Lower **Near** values allow the camera to display objects that are very close, at the cost of potential precision (Z-fighting) issues in the distance. Higher **Far** values allow the camera to see further away, also at the cost of potential precision (Z-fighting) issues in the distance.

The default focal length of 35 mm corresponds to a wide angle lens. It still results in a field of view that is noticeably narrower compared to the default "practical" vertical FOV of 75 degrees. This is because non-gaming use cases such as filmmaking and photography favor using a narrower field of view for a more cinematic appearance.

Common focal length values used in filmmaking and photography are:

- **Fisheye (ultrawide angle):** Below 15 mm. Nearly no depth of field visible.
- **Wide angle:** Between 15 mm and 50 mm. Reduced depth of field.
- **Standard:** Between 50 mm and 100 mm. Standard depth of field.
- **Telephoto:** Greater than 100 mm. Increased depth of field.

Like when using the **Keep Height** aspect mode, the effective field of view depends on the viewport's aspect ratio, with wider aspect ratios automatically resulting in a wider *horizontal* field of view.

Automatic exposure adjustment based on the camera's average brightness level can also be enabled in the **Auto Exposure** section, with the following properties:

- **Min Sensitivity:** The darkest brightness the camera is allowed to get to, measured in EV100.
- **Max Sensitivity:** The brightest the camera is allowed to get to, measured in EV100.
- **Speed:** The speed of the auto exposure effect. Affects the time needed for the camera to perform auto exposure. Higher values allow for faster transitions, but the resulting adjustments may look distracting depending on the scene.
- **Scale:** The scale of the auto exposure effect. Affects the intensity of auto exposure.

EV100 is an exposure value (EV) measured at an ISO sensitivity of 100. See this table [https://en.wikipedia.org/wiki/Exposure_value#Tabulated_exposure_values] for common EV100 values found in real life.

Particle systems (3D)

This section of the tutorial covers (3D) GPU-accelerated particle systems. Most of the things discussed here apply to CPU particles as well.

Introduction

You can use particle systems to simulate complex physical effects like fire, sparks, smoke, magical effects, and many more. They are very well suited for creating dynamic and organic behavior and adding "life" to your scenes.

The idea is that a particle is emitted at a fixed interval and with a fixed lifetime. During its lifetime, every particle will have the same base behavior. What makes each particle different from the others and creates the organic look is the randomness that you can add to most of its parameters and behaviors.

Every particle system you create in Godot consists of two main parts: particles and emitters.

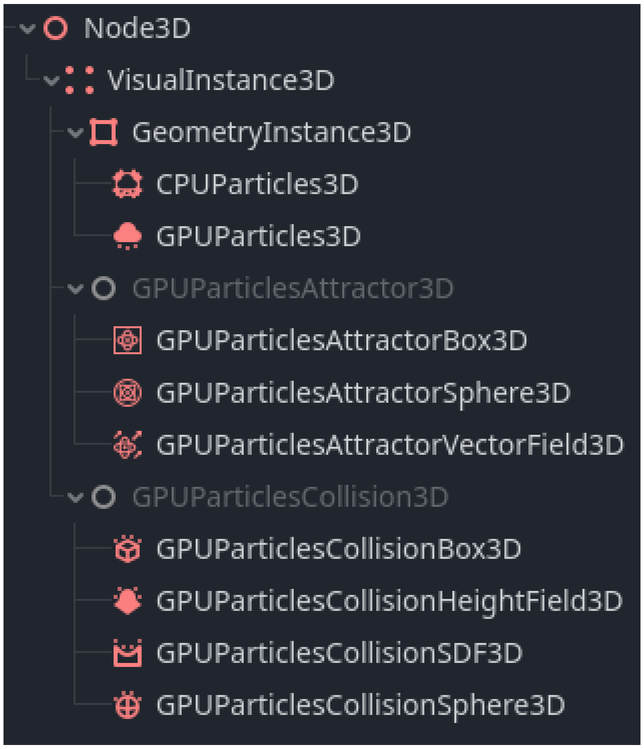
Particles

A particle is the visible part of a particle system. It's what you see on the screen when a particle system is active: The tiny specks of dust, the flames of a fire, the glowing orbs of a magical effect. You can have anywhere between a couple hundred and tens of thousands of particles in a single system. You can randomize a particle's size, its speed and movement direction, and change its color over the course of its lifetime. When you think of a fire, you can think of all the little embers flying away from it as individual particles.

Emitters

An emitter is what's creating the particles. Emitters are usually not visible, but they can have a shape. That shape controls where and how particles are spawned, for example whether they should fill a room like dust or shoot away from a single point like a fountain. Going back to the fire example, an emitter would be the heat at the center of the fire that creates the embers and the flames.

Node overview


Node3D
VisualInstance3D
GeometryInstance3D
CPUParticles3D
GPUParticles3D
GPUParticlesAttractor3D
GPUParticlesAttractorBox3D
GPUParticlesAttractorSphere3D
GPUParticlesAttractorVectorField3D
GPUParticlesCollision3D
GPUParticlesCollisionBox3D
GPUParticlesCollisionHeightField3D
GPUParticlesCollisionSDF3D
GPUParticlesCollisionSphere3D

All 3D particle nodes available in Godot

There are two types of 3D particle systems in Godot: GPUParticles3D, which are processed on the GPU, and CPUParticles3D, which are processed on the CPU.

CPU particle systems are less flexible than their GPU counterpart, but they work on a wider range of hardware and provide better support for older devices and mobile phones. Because they are processed on the CPU, they are not as performant as GPU particle systems and can't render as many individual particles. In addition they currently do not have all the available options GPU particles have for control.

GPU particle systems run on the GPU and can render hundreds of thousands of particles on modern hardware. You can write custom particle shaders for them, which makes them very flexible. You can also make them interact with the environment by using attractor and collision nodes.

There are three particle attractor nodes: GPUParticlesAttractorBox3D, GPUParticlesAttractorSphere3D, and GPUParticlesAttractorVectorField3D. An attractor node applies a force to all particles in its reach and pulls them closer or pushes them away based on the direction of that force.

There are several particle collision nodes. GPUParticlesCollisionBox3D and GPUParticlesCollisionSphere3D are the simple ones. You can use them to create basic shapes like boxes, a floor, or a wall that particles collide with. The other two nodes provide more complex collision behavior. The GPUParticlesCollisionSDF3D is useful when you want indoor scenes to collide with particles without having to create all the individual box and sphere colliders by hand. If you want particles to collide with large outdoor scenes, you would use the GPUParticlesCollisionHeightField3D node. It creates a heightmap of your world and the objects in it and uses that for large-scale particle collisions.

Basic usage

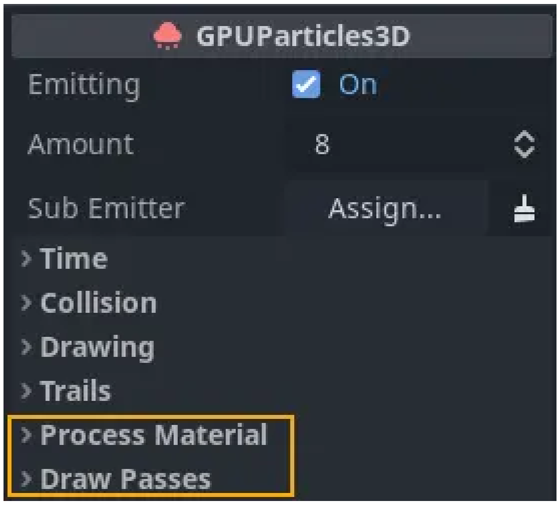
- Creating a 3D particle system
- 3D Particle system properties
- Process material properties

- Particle flags
- Spawn
- Accelerations
- Display
- Collision
- Sub-emitter

Advanced topics

- Particle sub-emitters
- 3D Particle trails
- Particle turbulence
- 3D Particle attractors
- 3D Particle collisions
- Complex emission shapes

Creating a 3D particle system

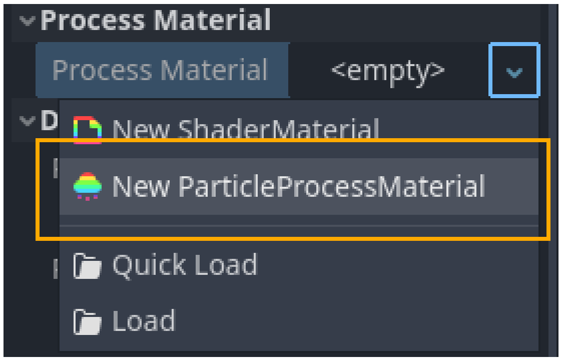

GPUParticles3D
Emitting
☑ On
Amount
8
Sub Emitter
Assign...
Time
Collision
Drawing
Trails
Process Material
Draw Passes

Required particle node properties

To get started with particles, the first thing we need to do is add a `GPUParticles3D` node to the scene. Before we can actually see any particles, we have to set up two parameters on the node: the `Process Material` and at least one `Draw Pass`.

The process material

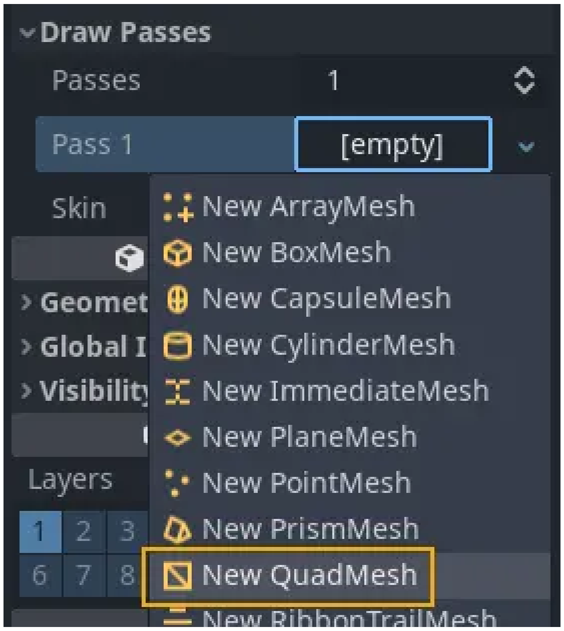
To add a process material to your particles node, go to `Process Material` in the inspector panel. Click on the box next to `Process Material` and from the dropdown menu select `New ParticleProcessMaterial`.


Process Material
Process Material
<empty>
New ShaderMaterial
New ParticleProcessMaterial
Quick Load
Load

Creating a process material

ParticleProcessMaterial is a special kind of material. We don't use it to draw any objects. We use it to update particle data and behavior on the GPU instead of the CPU, which comes with a massive performance boost. A click on the newly added material displays a long list of properties that you can set to control each particle's behavior.

Draw passes


Draw Passes
Passes
1
Pass 1
[empty]
Skin
Geomet
Global I
Visibility
Layers
1 2 3
6 7 8
New ArrayMesh
New BoxMesh
New CapsuleMesh
New CylinderMesh
New ImmediateMesh
New PlaneMesh
New PointMesh
New PrismMesh
New QuadMesh
New RibbonTrailMesh

At least one draw pass is required

In order to render any particles, at least one draw pass needs to be defined. To do that, go to `Draw Passes` in the inspector panel. Click on the box next to `Pass 1` and select `New QuadMesh` from the dropdown menu. After that, click on the mesh and set its `Size` to 0.1 for both `x` and `y`. Reducing the mesh's size makes it a little easier to tell the individual particle meshes apart at this stage.

You can use up to 4 draw passes per particle system. Each pass can render a different mesh with its own unique material. All draw passes use the data that is computed by the process material, which is an efficient method for composing complex effects: Compute particle behavior once and feed it to multiple render passes.

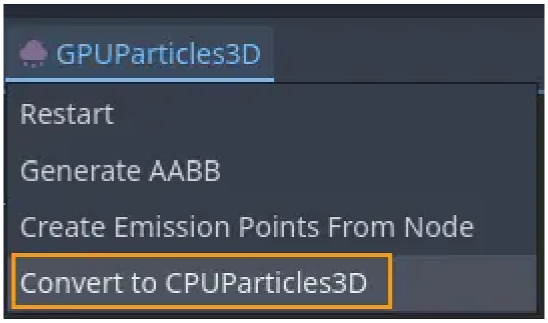
tutorials/3d/particles/img/particle_two_draw_passes.png

Using multiple draw passes: yellow rectangles (pass1) and blue spheres (pass 2)

If you followed the steps above, your particle system should now be emitting particles in a waterfall-like fashion, making them move downwards and disappear after a few seconds. This is the foundation for all particle effects. Take a look at the documentation for particle and particle material properties to learn how to make particle effects more interesting.

tutorials/3d/particles/img/particle_basic_system.png

Particle conversion


GPUParticles3D
Restart
Generate AABB
Create Emission Points From Node
Convert to CPUParticles3D

Turning GPU into CPU particles

You can convert GPU particles to CPU particles at any time using the entry in the viewport menu. When you do so, keep in mind that not every feature of GPU particles is available for CPU particles, so the resulting particle system will look and behave differently from the original.

You can also convert CPU particles to GPU particles if you no longer need to use CPU particles. This is also done from the viewport menu.

Some of the most notable features that are lost during the conversion include:

- multiple draw passes
- turbulence
- sub-emitters
- trails
- attractors
- collision

You also lose the following properties:

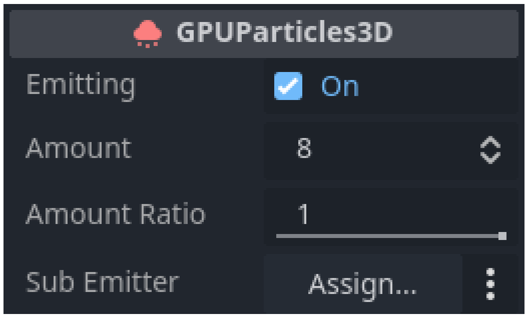
- `Amount Ratio`
- `Interp to End`
- `Damping as Friction`
- `Emission Shape Offset`
- `Emission Shape Scale`

- `Inherit Velocity Ratio`
- `Velocity Pivot`
- `Directional Velocity`
- `Radial Velocity`
- `Velocity Limit`
- `Scale Over Velocity`

Converting GPU particles to CPU particles can become necessary when you want to release a game on older devices that don't support modern graphics APIs.

3D Particle system properties

Emitter properties


GPUParticles3D

Emitting	☑ On
Amount	8
Amount Ratio	1
Sub Emitter	Assign...

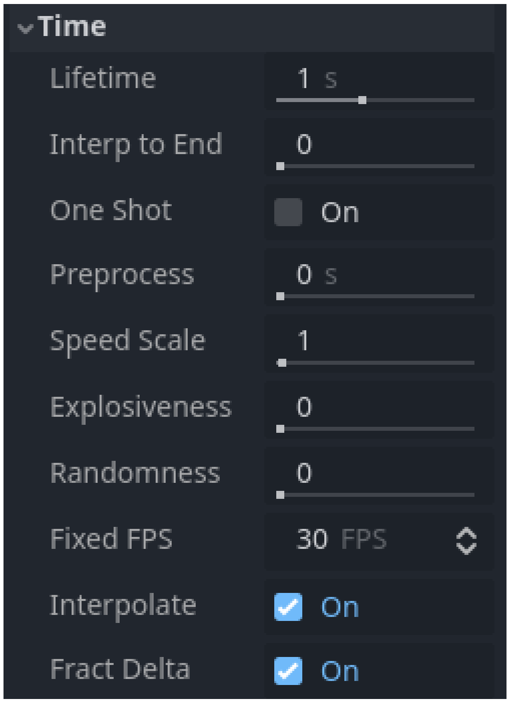
The checkbox next to the `Emitting` property activates and deactivates the particle system. Particles will only be processed and rendered if the box is checked. You can set this property at runtime if you want to activate or deactivate particle systems dynamically.

The `Amount` property controls the maximum number of particles visible at any given time. Increase the value to spawn more particles at the cost of performance.

The `Amount Ratio` property is the ratio of particles compared to the amount that will be emitted. If it's less than `1.0`, the amount of particles emitted through the lifetime will be the `Amount` * `Amount Ratio`. Changing this value while emitted doesn't affect already created particles and doesn't cause the particle system to restart. It's useful for making effects where the number of emitted particles varies over time.

You can set another particle node as a `Sub Emitter`, which will be spawned as a child of each particle. See the Sub-emitters section in this manual for a detailed explanation of how to add a sub-emitter to a particle system.

Time properties



Time

Lifetime 1 s

Interp to End 0

One Shot ☐ On

Preprocess 0 s

Speed Scale 1

Explosiveness 0

Randomness 0

Fixed FPS 30 FPS

Interpolate ☑ On

Fract Delta ☑ On

The `Lifetime` property controls how long each particle exists before it disappears again. It is measured in seconds. A lot of particle properties can be set to change over the particle's lifetime and blend smoothly from one value to another.

`Lifetime` and `Amount` are related. They determine the particle system's emission rate. Whenever you want to know how many particles are spawned per second, this is the formula you would use:

$$Particles per second = \frac{Amount}{Lifetime}$$

Example: Emitting 32 particles with a lifetime of 4 seconds each would mean the system emits 8 particles per second.

The `Interp to End` property causes all the particles in the node to interpolate towards the end of their lifetime.

If the checkbox next to the `One Shot` property is checked, the particle system will emit `amount` particles and then disable itself. It "runs" only once. This property is unchecked by default, so the system will keep emitting particles until it is disabled or destroyed manually. One-shot particles are a good fit for effects that react to a single event, like item pickups or splinters that burst away when a bullet hits a wall.

The `Preprocess` property is a way to fast-forward to a point in the middle of the particle system's lifetime and start rendering from there. It is measured in seconds. A value of `1` means that when the particle system starts, it will look as if it has been running for one second already.

This can be useful if you want the particle system to look like it has been active for a while even though it was just loaded into the scene. Consider the example below. Both particle systems simulate dust flying around in the area. With a preprocess value of `0`, there wouldn't be any dust for the first couple of seconds because the system has not yet emitted enough particles for the effect to become noticeable. This can be seen in the video on the left. Compare that to the video on the right where the particle system is preprocessed for `4` seconds. The dust is fully visible from the very beginning because we skipped the first four seconds of "setup" time.

tutorials/3d/particles/img/particle_preprocess.png

No preprocess (left) vs. 4 seconds of preprocess (right)

You can slow down or speed up the particle system with the `Speed Scale` property. This applies to processing the data as well as rendering the particles. Set it to `0` to pause the particle system completely or set it to something like `2` to make it move twice as fast.

tutorials/3d/particles/img/particle_speed_scale.png

Different speed scale values: 0.1 (left), 0.5 (middle), 1.0 (right)

The `Explosiveness` property controls whether particles are emitted sequentially or simultaneously. A value of `0` means that particles emit one after the other. A value of `1` means that all `amount` particles emit at the same time, giving the effect a more "explosive" appearance.

The `Randomness` property adds some randomness to the particle emission timing. When set to `0`, there is no randomness at all and the interval between the emission of one particle and the next is always the same: the particles are emitted at *regular* intervals. A `Randomness` value of `1` makes the interval completely random. You can use this property to break up some of the uniformity in your effects. When `Explosiveness` is set to `1`, this property has no effect.

Particles running at low FPS

Interpolation on (left) vs. off (right)

The `Fixed FPS` property limits how often the particle system is processed. This includes property updates as well as collision and attractors. This can improve performance a lot, especially in scenes that make heavy use of particle collision. Note that this does not change the speed at which particles move or rotate. You would use the `Speed Scale` property for that.

When you set `Fixed FPS` to very low values, you will notice that the particle animation starts to look choppy. This can sometimes be desired if it fits the art direction, but most of the time, you'll want particle systems to animate smoothly. That's what the `Interpolate` property does. It blends particle properties between updates so that even a particle system running at `10` FPS appears as smooth as running at `60`.

Note

When using particle collision, tunneling can occur if the particles move fast and colliders are thin. This can be remedied by increasing `Fixed FPS` (at a performance cost).

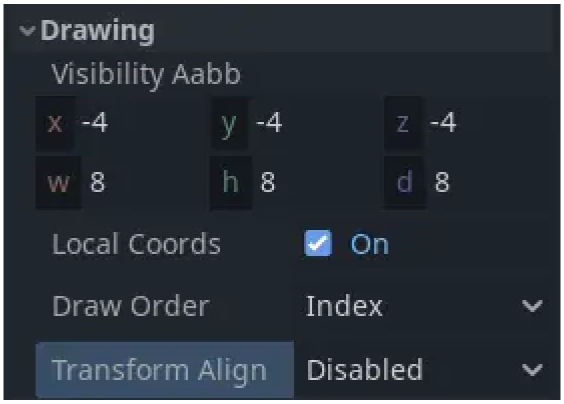
Collision properties

See also

Setting up particle collision requires following further steps described in 3D Particle collisions.

The `Base Size` property defines each particle's default collision size, which is used to check whether a particle is currently colliding with the environment. You would usually want this to be about the same size as the particle. It can make sense to increase this value for particles that are very small and move very fast to prevent them from clipping through the collision geometry.

Drawing properties



Drawing

Visibility Aabb

x -4 y -4 z -4

w 8 h 8 d 8

Local Coords ☑ On

Draw Order Index

Transform Align Disabled

The `Visibility AABB` property defines a box around the particle system's origin. As long as any part of this box is in the camera's field of view, the particle system is visible. As soon as it leaves the camera's field of view, the particle system stops being rendered at all. You can use this property to boost performance by keeping the box as small as possible.

One thing to keep in mind when you set a size for the `Visibility AABB` is that particles that are outside of its bounds disappear instantly when it leaves the camera's field of view. Particle collision will also not occur outside the `Visibility AABB`. While not technically a bug, this can have a negative effect on the visual experience.

When the `Local Coords` property is checked, all particle calculations use the local coordinate system to determine things like up and down, gravity, and movement direction. Up and down, for example, would follow the particle system's or its parent node's rotation. When the property is unchecked, the global world space is used for these calculations: Down will always be -Y in world space, regardless of the particle system's rotation.

tutorials/3d/particles/img/particle_coords.png

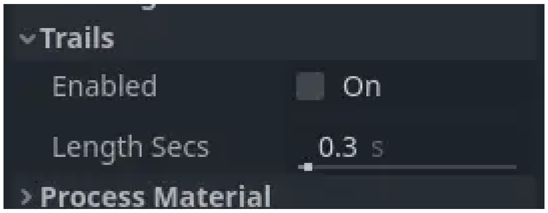
Local space coordinates (left) vs. world space coordinates (right)

The `Draw Order` property controls the order in which individual particles are drawn. `Index` means that they are drawn in the order of emission: particles that are spawned later are drawn on top of earlier ones. `Lifetime` means that they are drawn in the order of their remaining lifetime. `Reverse Lifetime` reverses the `Lifetime` draw order. `View Depth` means particles are drawn according to their distance from the camera: The ones closer to the camera on top of those farther away.

The `Transform Align` property controls the particle's default rotation. `Disabled` means they don't align in any particular way. Instead, their rotation is determined by the values set in the process material. `Z-`

`Billboard` means that the particles will always face the camera. This is similar to the `Billboard` property in the Standard Material. `Y to Velocity` means that each particle's Y-axis aligns with its movement direction. This can be useful for things like bullets or arrows, where you want particles to always point "forward". `Z-Billboard + Y to Velocity` combines the previous two modes. Each particle's Z-axis will point towards the camera while its Y-axis will align with their velocity.

Trail properties



Enabled ☐ On

Length Secs 0.3 s

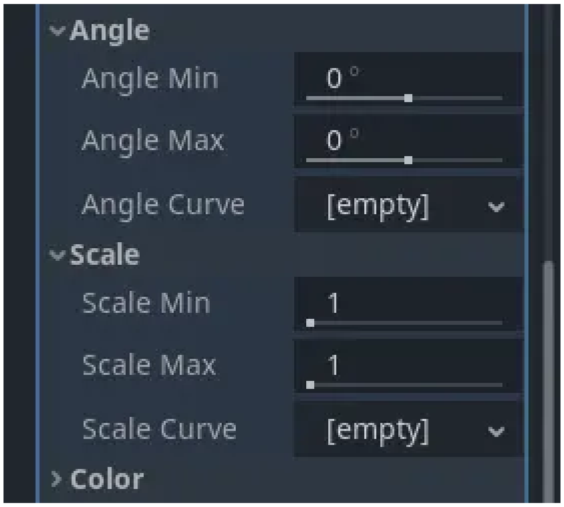
Particle trail properties

The `Enabled` property controls whether particles are rendered as trails. The box needs to be checked if you want to make use of particle trails.

The `Length Secs` property controls for how long a trail should be emitted. The longer this duration is, the longer the trail will be.

See the Particle trails section in this manual for a detailed explanation of how particle trails work and how to set them up.

Process material properties



Angle

Angle Min 0 °

Angle Max 0 °

Angle Curve [empty]

Scale

Scale Min 1

Scale Max 1

Scale Curve [empty]

Color

Min, max, and curve properties

The properties in this material control how particles behave and change over their lifetime. A lot of them have `Min`, `Max`, and `Curve` values that allow you to fine-tune their behavior. The relationship between these values is this: When a particle is spawned, the property is set with a random value between `Min` and `Max`. If `Min` and `Max` are the same, the value will always be the same for every particle. If the `Curve` is also set, the value of the property will be multiplied by the value of the curve at the current point in a particle's lifetime. Use the curve to change a property over the particle lifetime. Very complex behavior can be expressed this way.

Time

The `Lifetime Randomness` property controls how much randomness to apply to each particle's lifetime. A value of `0` means there is no randomness at all and all particles live for the same amount of time, set by the Lifetime property. A value of `1` means that a particle's lifetime is completely random within the range of [0.0, `Lifetime`].

Particle flags

The `Align Y` property aligns each particle's Y-axis with its velocity. Enabling this property is the same as setting the Transform Align property to `Y to Velocity`.

The `Rotate Y` property works with the properties in the Angle and Angular Velocity groups to control particle rotation. `Rotate Y` has to be enabled if you want to apply any rotation to particles. The exception to this is any particle that uses the Standard Material where the `Billboard` property is set to `Particle Billboard`. In that case, particles rotate even without `Rotate Y` enabled.

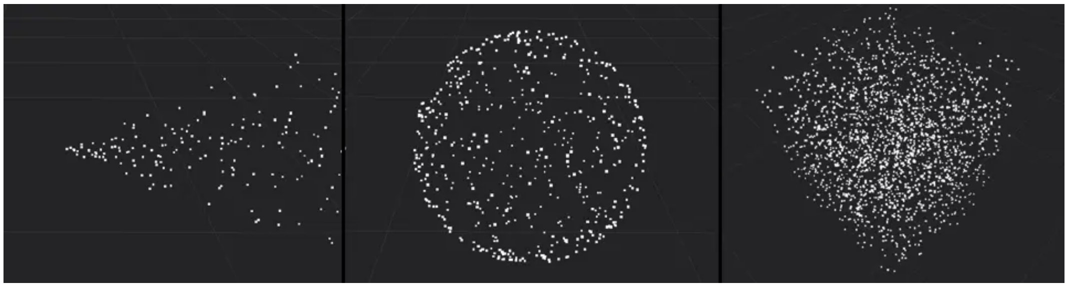
When the `Disable Z` property is enabled, particles will not move along the Z-axis. Whether that is going to be the particle system's local Z-axis or the world Z-axis is determined by the Local Coords property.

The `Damping as Friction` property changes the behavior of damping from a constant deceleration to a deceleration based on speed.

Spawn

Emission shape

Particles can emit from a single point in space or in a way that they fill out a shape. The `Shape` property controls that shape. `Point` is the default value. All particles emit from a single point in the center of the particle system. When set to `Sphere` or `Box`, particles emit in a way that they fill out a sphere or a box shape evenly. You have full control over the size of these shapes. `Sphere Surface` works like `Sphere`, but instead of filling it out, all particles spawn on the sphere's surface.



Particles emitting from a point (left), in a sphere (middle), and in a box (right)

A ring-shaped particle system

The `Ring` emission shape makes particles emit in the shape of a ring. You can control the ring's direction by changing the `Ring Axis` property. `Ring Height` controls the thickness of the ring along its axis. `Ring Radius` and `Ring Inner Radius` control how wide the ring is and how large the hole in the middle should be. The image shows a particle system with a radius of `2` and an inner radius of `1.5`, the axis points along the global Z-axis.

In addition to these relatively simple shapes, you can select the `Points` or `Directed Points` option to create highly complex emission shapes. See the Complex emission shapes section for a detailed explanation of how to set these up.

Angle

The `Angle` property controls a particle's starting rotation as described above. In order to have an actual effect on the particle, you have to enable one of two properties: Rotate Y rotates the particle around the particle system's Y-axis. The `Billboard` property in the Standard Material, if it is set to `Particle Billboard`, rotates the particle around the axis that points from the particle to the camera.

Direction

Note

The `Direction` property alone is not enough to see any particle movement. Whatever values you set here only take effect once velocity or acceleration properties are set, too.

The `Direction` property is a vector that controls each particle's direction of movement at the moment it is spawned. A value of `(X=1,Y=0,Z=0)` would make all particles move sideways along the X-axis. For something like a fountain where particles shoot out up in the air, a value of `(X=0,Y=1,Z=0)` would be a good starting point.

Different values for particle direction

Different direction values: Y-axis only (left), equal values for X and Y (middle), X and Y with gravity enabled (right)

After setting a direction, you will notice that all particles move in the same direction in a straight line. The `Spread` property adds some variation and randomness to each particle's direction. The higher the value, the stronger the deviation from the original path. A value of `0` means there is no spread at all while a value of `180` makes particles shoot out in every direction. You could use this for something like pieces of debris during an explosion effect.

> Different values for particle spread
>
> No spread (left), 45 degree angle (middle), full 180 degrees (right)

The `Flatness` property limits the spread along the Y-axis. A value of `0` means there is no limit and a value of `1` will eliminate all particle movement along the Y-axis. The particles will spread out completely "flat".

You won't see any actual movement until you also set some values for the velocity and acceleration properties below, so let's take a look at those next.

Initial velocity

While the `Direction` property controls a particle's movement direction, the `Initial Velocity` controls how fast it goes. It's separated into `Velocity Min` and `Velocity Max`, both set to `0` by default, which is why you don't see any movement initially. As soon as you set values for either of these properties as described above, the particles begin to move. The direction is multiplied by these values, so you can make particles move in the opposite direction by setting a negative velocity.

Accelerations

Gravity

The next few property groups work closely together to control particle movement and rotation. `Gravity` drags particles in the direction it points at, which is straight down at the strength of Earth's gravity by default. Gravity affects all particle movement. If your game uses physics and the world's gravity can change at runtime, you can use this property to keep the game's gravity in sync with particle gravity. A `Gravity` value of `(X=0,Y=0,Z=0)` means no particle will ever move at all if none of the other movement properties are set.

Different values for particle gravity

Left: (X=0,Y=-9.8,Z=0), middle: (X=0,Y=9.8,Z=0), right: (X=4,Y=2,Z=0).

Angular velocity

`Angular Velocity` controls a particle's speed of rotation as described above. You can reverse the direction by using negative numbers for `Velocity Min` or `Velocity Max`. Like the Angle property, the rotation will only be visible if the Rotate Y flag is set or the `Particle Billboard` mode is selected in the Standard Material.

Note

The Damping property has no effect on the angular velocity.

Linear acceleration

A particle's velocity is a constant value: once it's set, it doesn't change and the particle will always move at the same speed. You can use the `Linear Accel` property to change the speed of movement over a particle's lifetime as described above. Positive values will speed up the particle and make it move faster. Negative values will slow it down until it stops and starts moving in the other direction.

Different values for particle linear acceleration

Negative (top) and positive (bottom) linear acceleration

It's important to keep in mind that when we change acceleration, we're not changing the velocity directly, we're changing the *change* in velocity. A value of `0` on the acceleration curve does not stop the particle's movement, it stops the change in the particle's movement. Whatever its velocity was at that moment, it will keep moving at that velocity until the acceleration is changed again.

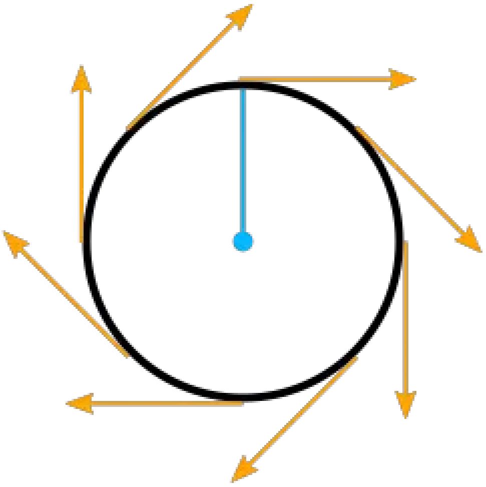
Radial acceleration

The `Radial Accel` property adds a gravity-like force to all particles, with the origin of that force at the particle system's current location. Negative values make particles move towards the center, like the force of gravity from a planet on objects in its orbit. Positive values make particles move away from the center.

Different values for particle radial acceleration

Negative (left) and positive (right) radial acceleration

Tangential acceleration



Tangents on a circle

This property adds particle acceleration in the direction of the tangent to a circle on the particle system's XZ-plane with the origin at the system's center and a radius the distance between each particle's current location and the system's center projected onto that plane.

Let's unpack that.

A tangent to a circle is a straight line that "touches" the circle in a right angle to the circle's radius at the touch point. A circle on the particle system's XZ-plane is the circle that you see when you look straight down at the particle system from above.

Tangential acceleration from above

Tangential acceleration from above

`Tangential Accel` is always limited to that plane and never move particles along the system's Y-axis. A particle's location is enough to define such a circle where the distance to the system's center is the radius if we ignore the vector's Y component.

The `Tangential Accel` property will make particles orbit the particle system's center, but the radius will increase constantly. Viewed from above, particles will move away from the center in a spiral. Negative values reverse the direction.

Damping

The `Damping` property gradually stops all movement. Each frame, a particle's movement is slowed down a little unless the total acceleration is greater than the damping effect. If it isn't, the particle will keep slowing down until it doesn't move at all. The greater the value, the less time it takes to bring particles to a complete halt.

Attractor interaction

If you want the particle system to interact with particle attractors, you have to check the `Enabled` property. When it is disabled, the particle system ignores all particle attractors.

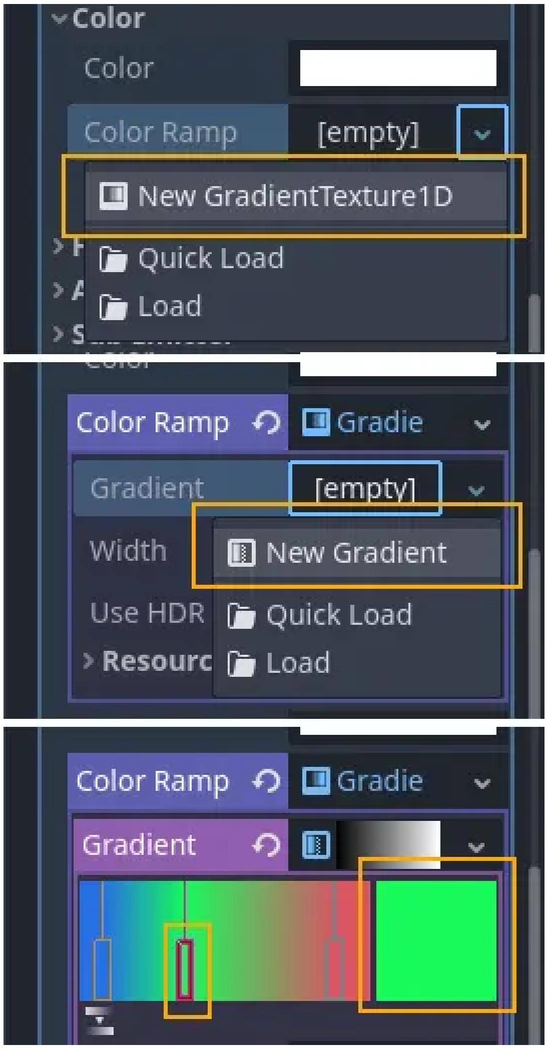
Display

Scale

`Scale` controls a particle's size as described above. You can set different values for `Scale Min` and `Scale Max` to randomize each particle's size. Negative values are not allowed, so you won't be able to flip particles with this property. If you emit particles as billboards, the `Keep Size` property on the Standard Material in your draw passes has to be enabled for any scaling to have an effect.

Color

The `Color` property controls a particle's initial color. It will have an effect only after the `Use As Albedo` property in the `Vertex Color` group of the [Standard Material](#) is enabled. This property is multiplied with color coming from the particle material's own `Color` or `Texture` property.


Color
Color
Color Ramp
[empty]
New GradientTexture1D
Quick Load
Load
Color Ramp
Gradie
Gradient
[empty]
Width
New Gradient
Use HDR
Quick Load
Resourc
Load
Color Ramp
Gradie
Gradient

Setting up a color ramp

There are two `Ramp` properties in the `Color` group. These allow you to define a range of colors that are used to set the particle's color. The

`Color Ramp` property changes a particle's color over the course of its lifetime. It moves through the entire range of colors you defined. The `Color Initial Ramp` property selects the particle's initial color from a random position on the color ramp.

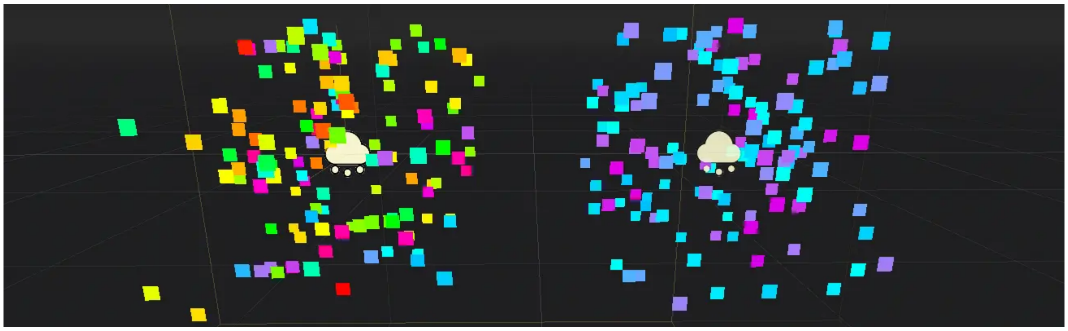
To set up a color ramp, click on the box next to the property name and from the dropdown menu select `New GradientTexture1D`. Click on the box again to open the texture's details. Find the `Gradient` property, click on the box next to it and select `New Gradient`. Click on that box again and you will see a color range. Click anywhere on that range to insert a new marker. You can move the marker with the mouse and delete it by clicking the right mouse button. When a marker is selected, you can use the color picker next to the range to change its color.

Hue variation

Like the `Color` property, `Hue Variation` controls a particle's color, but in a different way. It does so not by setting color values directly, but by *shifting the color's hue*.

Hue describes a color's pigment: red, orange, yellow, green and so on. It does not tell you anything about how bright or how saturated the color is. The `Hue Variation` property controls the range of available hues as described above.

It works on top of the particle's current color. The values you set for `Variation Min` and `Variation Max` control how far the hue is allowed to shift in either direction. A higher value leads to more color variation while a low value limits the available colors to the closest neighbors of the original color.

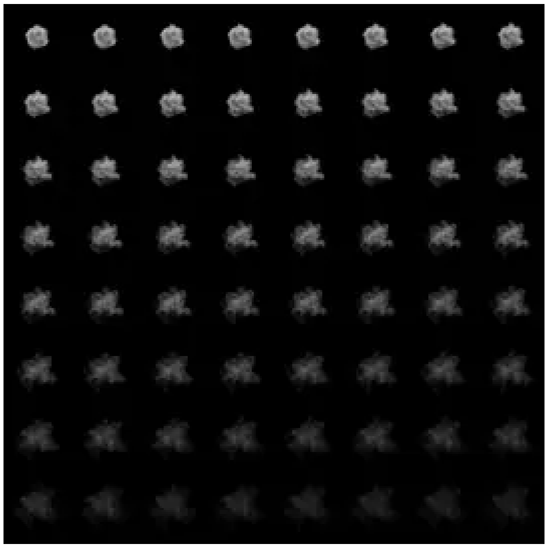


Different values for hue variation, both times with blue as base color: 0.6 (left) and 0.1 (right)

Animation

The `Animation` property group controls the behavior of sprite sheet animations in the particle's Standard Material. The `Min`, `Max`, and `Curve` values work as described above.

An animated sprite sheet is a texture that contains several smaller images aligned on a grid. The images are shown one after the other so fast that they combine to play a short animation, like a flipbook. You can use them for animated particles like smoke or fire. These are the steps to create an animated particle system:



An 8x8 animated smoke sprite sheet

1. Import a sprite sheet texture into the engine. If you don't have one at hand, you can download the **`high-res version of the example image.`**
2. Set up a particle system with at least one draw pass and assign a `Standard Material` to the mesh in that draw pass.
3. Assign the sprite sheet to the `Texture` property in the `Albedo` group
4. Set the material's `Billboard` property to `Particle Billboard`. Doing so makes the `Particles Anim` group available in the material.
5. Set `H Frames` to the number of columns and `V Frames` to the number of rows in the sprite sheet.
6. Check `Loop` if you want the animation to keep repeating.

That's it for the Standard Material. You won't see any animation right away. This is where the `Animation` properties come in. The `Speed` properties control how fast the sprite sheet animates. Set `Speed Min` and `Speed Max` to `1` and you should see the animation playing. The `Offset` properties control where the animation starts on a newly spawned particle. By default, it will always be the first image in the sequence. You can add some variety by changing `Offset Min` and `Offset Max` to randomize the starting position.

Animated particles

Three different particle systems using the same smoke sprite sheet

Depending on how many images your sprite sheet contains and for how long your particle is alive, the animation might not look smooth. The relationship between particle lifetime, animation speed, and number of images in the sprite sheet is this:

Note

At an animation speed of `1.0`, the animation will reach the last image in the sequence just as the particle's lifetime ends.

$$Animation\ FPS = \frac{Number\ of\ images}{Lifetime}$$

If your sprite sheet contains 64 (8x8) images and the particle's lifetime is set to `1 second`, the animation will be very smooth at **64 FPS** (1 second / 64 images). if the lifetime is set to `2 seconds`, it will still be fairly smooth at **32 FPS**. But if the particle is alive for `8 seconds`, the animation will be visibly choppy at **8 FPS**. In order to make the animation smooth again, you need to increase the animation speed to something like `3` to reach an acceptable framerate.

Animated particles lifetimes

The same particle system at different lifetimes: 1 second (left), 2 seconds (middle), 8 seconds (right)

Note that the GPUParticles3D node's **Fixed FPS** also affects animation playback. For smooth animation playback, it's recommended to set it to 0 so that the particle is simulated on every rendered frame. If this is not an option for your use case, set **Fixed FPS** to be equal to the effective framerate used by the flipbook animation (see above for the formula).

Turbulence

Turbulence adds noise to particle movement, creating interesting and lively patterns. Check the box next to the `Enabled` property to activate it. A number of new properties show up that control the movement speed, noise pattern and overall influence on the particle system. You can find a detailed explanation of these in the section on particle turbulence.

Collision

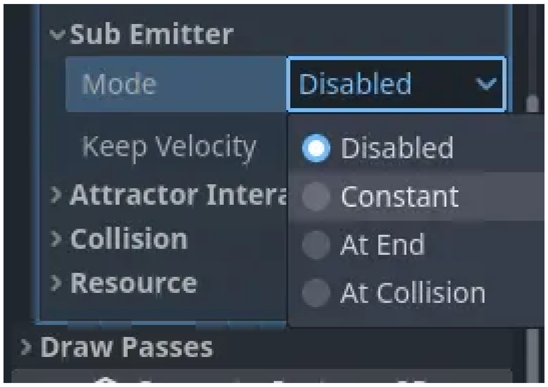
The `Mode` property controls how and if emitters collide with particle collision nodes. Set it to `Disabled` to disable any collision for this particle system. Set it to `Hide On Contact` if you want particles to disappear as soon as they collide. Set it to `Constant` to make particles collide and bounce around. You will see two new properties appear in the inspector. They control how particles behave during collision events.

A high `Friction` value will reduce sliding along surfaces. This is especially helpful if particles collide with sloped surfaces and you want them to stay in place instead of sliding all the way to the bottom, like snow falling on a mountain. A high `Bounce` value will make particles bounce off surfaces they collide with, like rubber balls on a solid floor.

If the `Use Scale` property is enabled, the collision base size is multiplied by the particle's current scale. You can use this to make sure that the rendered size and the collision size match for particles with random scale or scale that varies over time.

You can learn more about particle collisions in the Collisions section in this manual.

Sub-emitter


Sub Emitter
Mode
Disabled
Keep Velocity
Attractor Intera
Collision
Resource
● Disabled
○ Constant
○ At End
○ At Collision
Draw Passes

The available sub-emitter modes

The `Mode` property controls how and when sub-emitters are spawned. Set it to `Disabled` and no sub-emitters will ever be spawned. Set it to `Constant` to make sub-emitters spawn continuously at a constant rate. The `Frequency` property controls how often that happens within the span of one second. Set the mode to `At End` to make the sub-emitter spawn at the end of the parent particle's lifetime, right before it is destroyed. The `Amount At End` property controls how many sub-emitters will be spawned. Set the mode to `At Collision` to make sub-emitters spawn when a particle collides with the environment. The `Amount At Collision` property controls how many sub-emitters will be spawned.

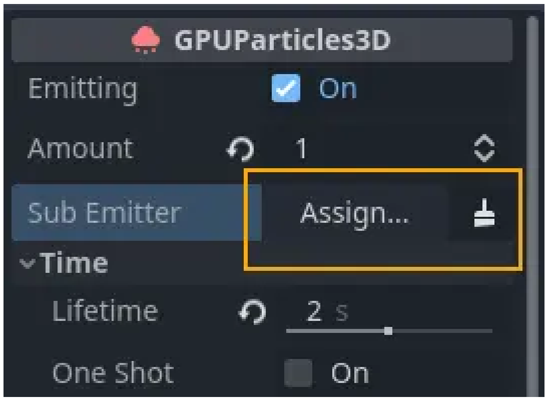
When the `Keep Velocity` property is enabled, the newly spawned sub-emitter starts off with the parent particle's velocity at the time the sub-emitter is created.

See the Sub-emitters section in this manual for a detailed explanation of how to add a sub-emitter to a particle system.

Particle sub-emitters

Chained sub-emitters

Sometimes a visual effect cannot be created with a single particle system alone. Sometimes a particle system needs to be spawned as a response to something that happens in another particle system. Fireworks are a good example of that. They usually consist of several stages of explosions that happen in sequence. Sub-emitters are a good way to achieve this kind of effect.



GPUParticles3D

Emitting ☑ On

Amount 1

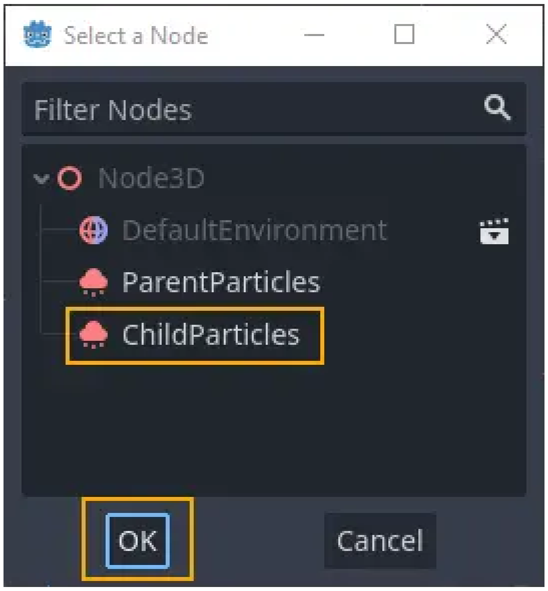
Sub Emitter Assign...

Time

Lifetime 2 s

One Shot ☐ On

Click to assign a sub-emitter...



...and select one from the scene

A sub-emitter is a particle system that spawns as a child of another particle system. You can add sub-emitters to sub-emitters, chaining particle effects as deep as you like.

To create a sub-emitter, you need at least two particle systems in the same scene. One of them will be the parent and one will be set as the child. Find the `Sub Emitter` property on the parent and click the box next to it to assign the sub-emitter. You will see a list of available particle systems in the scene. Select one and click the confirmation button.

Particle systems from instanced scenes can be set as sub-emitters too, as long as the `Editable Children` property is enabled on the instanced scene. This also works the other way around: You can assign a sub-emitter to a particle system in an instanced scene, even one coming from a different instanced scene.

Note

When you set a particle system as the sub-emitter of another, the system stops emitting, even if the `Emitting` property was checked.

Don't worry, it didn't break. This happens to every particle system as soon as it becomes a sub-emitter. You also won't be able to re-enable the property as long as the particle system is used as a sub-emitter.

Warning

Even though the parent particle system can be selected from the list of available particle systems, a particle system which is its own sub-emitter does not work in Godot. It will simply not spawn. The same is true for any other kind of recursive or self-referential sub-emitter setup.

Emitter mode

When you assign a sub-emitter, you don't see it spawn right away. Emitting is disabled by default and needs to be enabled first. Set the `Mode` property in the `Sub Emitter` group of the ParticleProcessMaterial to something other than `Disabled`.

The emitter mode also determines how many sub-emitter particles are spawned. `Constant` spawns a single particle at a frequency set by the `Frequency` property. For `At End` and `At Collision` you can set the amount directly with the `Amount At End` and the `Amount At Collision` properties.

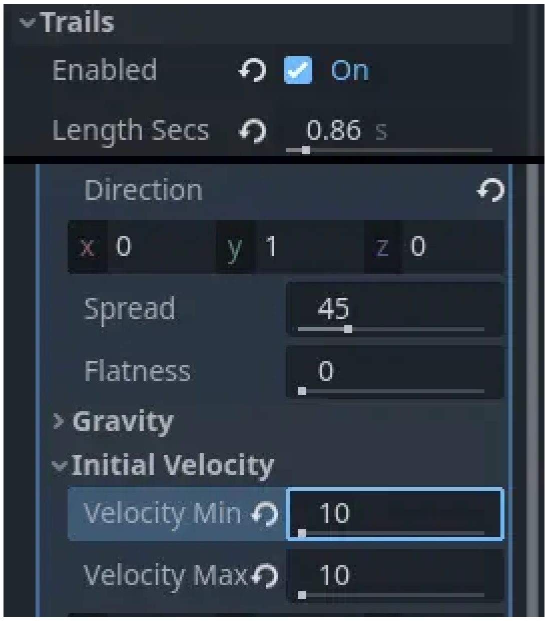
Limitations

One thing to keep in mind is that the total number of active particles from the sub-emitter is always capped by the `Amount` property on the sub-emitter particle system. If you find that there are not enough particles spawned from the sub-emitter, you might have to increase the amount in the particle system.

Some emitter properties are ignored when a particle system is spawned as a sub-emitter. The `Explosiveness` property, for example, has no effect. Depending on the emitter mode, the particles are either spawned sequentially at fixed intervals or explosively all at once.

3D Particle trails

Particle trails



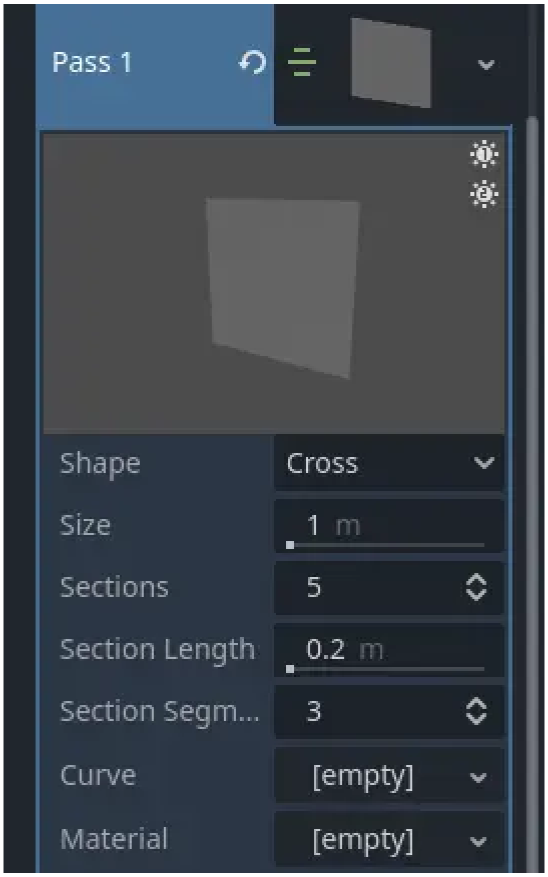
Trails
Enabled ☑ On
Length Secs 0.86 s
Direction
x 0 y 1 z 0
Spread 45
Flatness 0
Gravity
Initial Velocity
Velocity Min 10
Velocity Max 10

Setting up particle trails

Godot provides several types of trails you can add to a particle system. Before you can work with trails, you need to set up a couple of parameters first. Create a new particle system and assign a process material as described before. In the `Trails` group of the particle system, check the box next to `Enabled` and increase the emission duration by setting `Lifetime` to something like `0.8`. On the process material, set `Direction` to `(X=0,Y=1.0,Z=0)` and `Initial Velocity` to `10.0` for both `Min` and `Max`.

The only thing that's still missing is a mesh for the draw pass. The type of mesh that you set here controls what kind of particle trail you will end up with.

Ribbon trails



Shape: Cross
Size: 1 m
Sections: 5
Section Length: 0.2 m
Section Segm...: 3
Curve: [empty]
Material: [empty]

Important ribbon mesh parameters

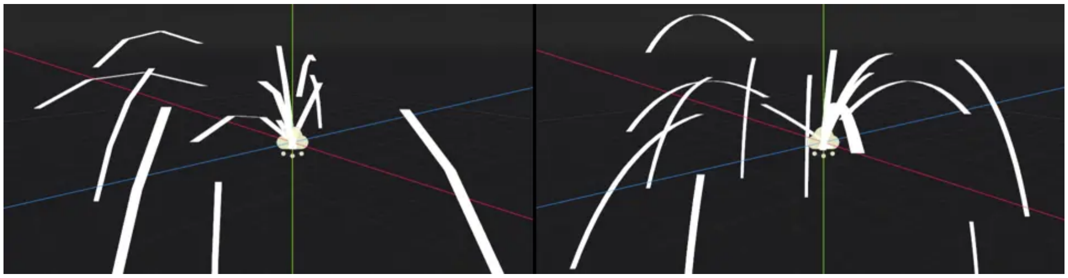
The simplest type of particle trail is the ribbon trail. Navigate to the `Draw Passes` section and select `New RibbonTrailMesh` from the options for `Pass 1`. A RibbonTrailMesh is a simple quad that is divided into sections and then stretched and repeated along those sections.

Assign a new Standard Material to the `Material` property and enable `Use Particle Trails` in the `Transform` property group. The particles should now be emitting in trails.

You have two options for the ribbon mesh `Shape` parameter. `Cross` creates two perpendicular quads, making the particle trail a little more three-dimensional. This really only makes sense if you don't draw the trails in `Particle Billboard` mode and helps when looking at the particles from different angles. The `Flat` option limits the mesh to a single quad and works best with billboard particles.

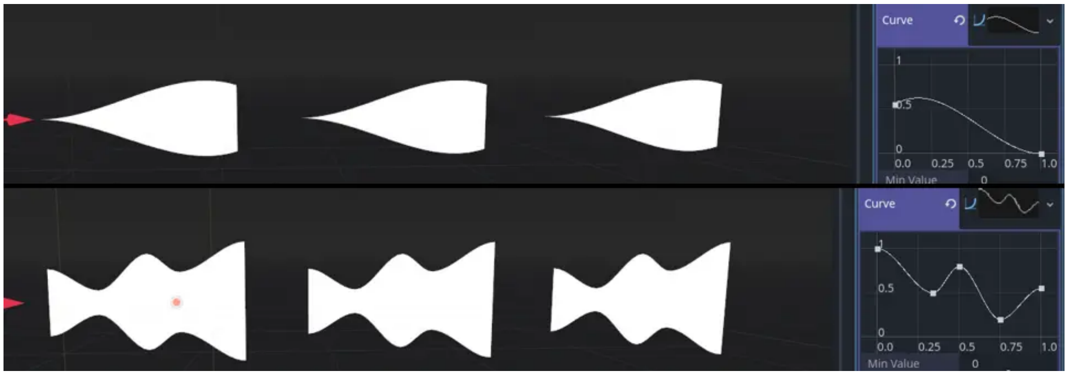
The `Size` parameter controls the trail's width. Use it to make trails wider or more narrow.

`Sections`, `Section Length` and `Section Segments` all work together to control how smooth the particle trail looks. When a particle trail does not travel in a straight line, the more sections it has the smoother it looks as it bends and swirls. `Section Length` controls the length of each section. Multiply this value by the number of sections to know the trail's total length.



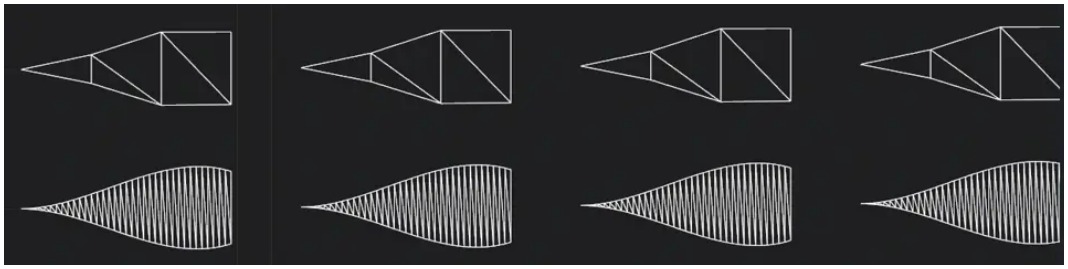
3 sections, 1m section length (left) vs. 12 sections, 0.25m section length (right). Notice how the total length of the trails stays the same.

The `Section Segments` parameter further subdivides each section into segments. It has no effect on the smoothness of the trail's sections, though. Instead, it controls the smoothness of the particle trail's overall shape. The `Curve` property defines this shape. Click the box next to `Curve` and assign or create a new curve. The trail will be shaped just like the curve with the curve's value at `0.0` at the trail's head and the curve's value at `1.0` at the trail's tail.


Curve
1
0.5
0
0.0
0.25
0.5
0.75
1.0
Min Value
0
Curve
1
0.5
0
0.0
0.25
0.5
0.75
1.0
Min Value
0

Particle trails shaped by different curves. The trails move from left to right.

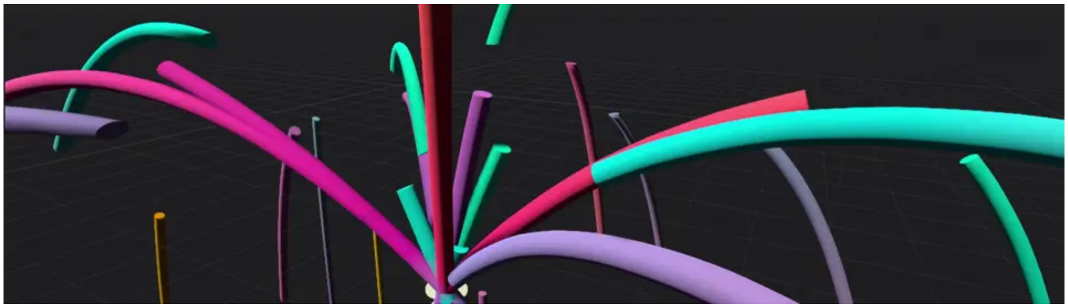
Depending on the complexity of the curve, the particle trail's shape will not look very smooth when the number of sections is low. This is where the `Section Segments` property comes in. Increasing the amount of section segments adds more vertices to the trail's sides so that it can follow the curve more closely.



Particle trail shape smoothness: 1 segment per section (top), 12 segments per section (bottom)

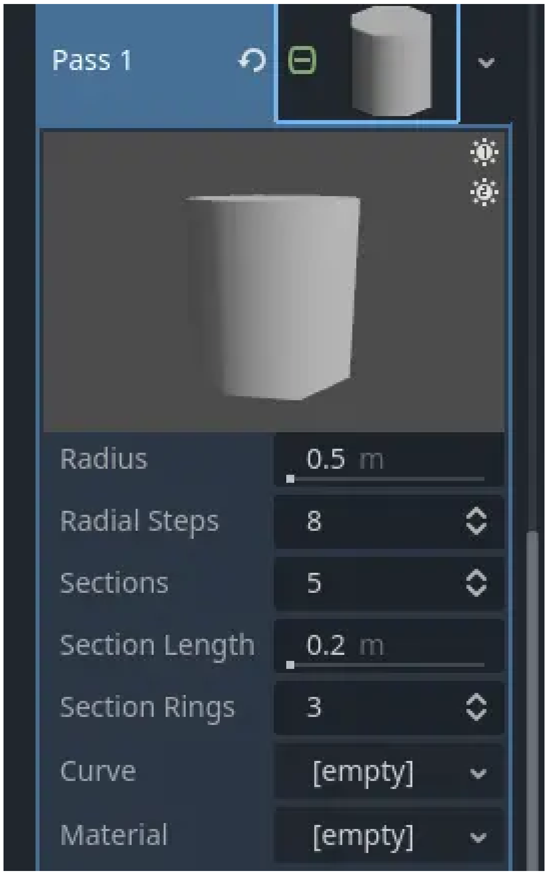
Tube trails

Tube trails share a lot of their properties with ribbon trails. The big difference between them is that tube trails emit cylindrical meshes instead of quads.



Tube trails emit cylindrical particles

To create a tube trail, navigate to the `Draw Passes` section and select `New TubeTrailMesh` from the options for `Pass 1`. A [TubeTrailMesh]() is a cylinder that is divided into sections and then stretched and repeated along those sections. Assign a new [Standard Material]() to the `Material` property and enable `Use Particle Trails` in the `Transform` property group. The particles should now be emitting in long, cylindrical trails.



Pass 1

Radius 0.5 m

Radial Steps 8

Sections 5

Section Length 0.2 m

Section Rings 3

Curve [empty]

Material [empty]

Important tube mesh parameters

The `Radius` and `Radial Steps` properties are to tube trails what `Size` is to ribbon trails. `Radius` defines the radius of the tube and increases or decreases its overall size. `Radial Steps` controls the number of sides around the tube's circumference. A higher value increases the resolution of the tube's cap.

`Sections` and `Section Length` work the same for tube trails and ribbon trails. They control how smooth the tube trail looks when it is bending and twisting instead of moving in a straight line. Increasing the number of sections will make it look smoother. Change the `Section Length` property to change the length of each section and with it the total length of the trail. `Section Rings` is the tube equivalent of the `Section Segments` property for ribbons. It subdivides the sections and adds more geometry to the tube to better fit the custom shape defined in the `Curve` property.

You can shape tube trails with curves, just as you can with ribbon trails. Click the box next to the `Curve` property and assign or create a new curve. The trail will be shaped like the curve with the curve's value at `0.0` at the trail's head and the curve's value at `1.0` at the trail's tail.

Particle tubes

Particle tube trails with a custom curve shape: 4 radial steps, 3 sections, 1 section ring (left), 12 radial steps, 9 sections, 3 section rings (right)

An important property you might want to set is `Transform Align` in the particle system's `Drawing` group. If you leave it as is, the tubes will not preserve volume; they flatten out as they move because their Y-axis keeps pointing up even as they change direction. This can cause a lot of rendering artifacts. Set the property to `Y to`

`Velocity` instead and each particle trail keeps its Y-axis aligned along the direction of its movement.

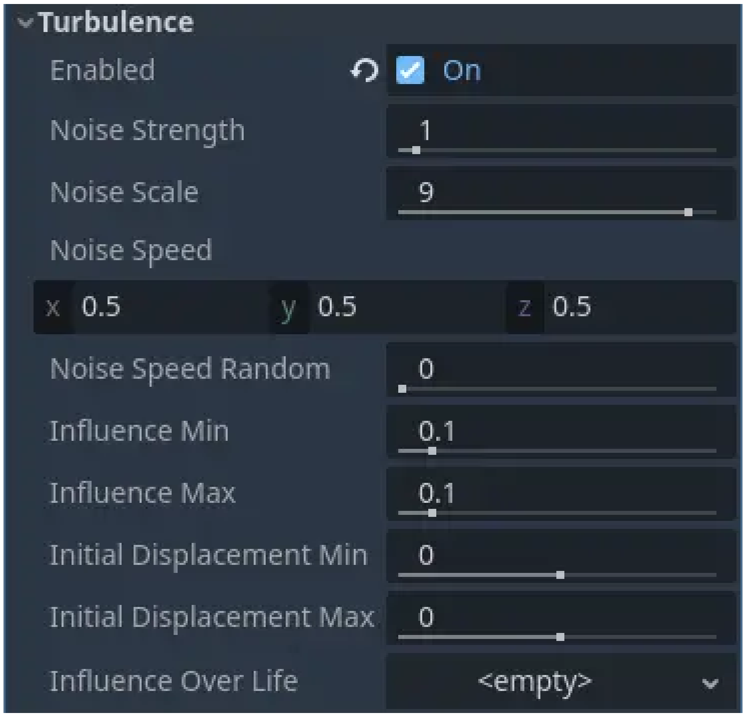
Particle tubes aligned

Particle tube trails without alignment (left) and with Y-axis aligned to velocity (right)

Particle turbulence

Particle turbulence

Turbulence uses a noise texture to add variation and interesting patterns to particle movement. It can be combined with particle attractors and collision nodes to create even more complex looking behavior.



Turbulence

Enabled: ☑ On

Noise Strength: 1

Noise Scale: 9

Noise Speed: x 0.5 y 0.5 z 0.5

Noise Speed Random: 0

Influence Min: 0.1

Influence Max: 0.1

Initial Displacement Min: 0

Initial Displacement Max: 0

Influence Over Life: <empty>

Particle turbulence properties

There are two things you have to do before turbulence has any effect on a particle system. First you must add movement to the particle system. Turbulence modifies a particle's movement direction and speed, but it doesn't create any. It is enough to give the particle system some gravity, but you can just as well create a number of attractors if you want the particles to follow a more complex movement path. Second, you need to enable turbulence in the particle process material. Once enabled, you have access to all the turbulence properties.

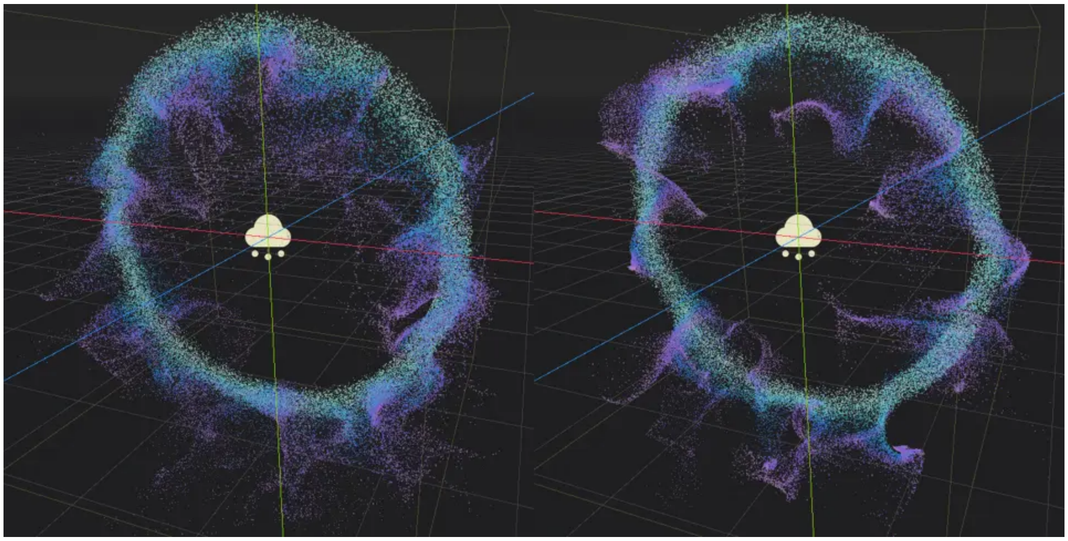
Warning

Turbulence makes use of 3D noise, which has a high performance cost on the GPU. Only enable turbulence on a few particle systems on screen at most. Using turbulence is not recommended when targeting mobile/web platforms.

Noise properties

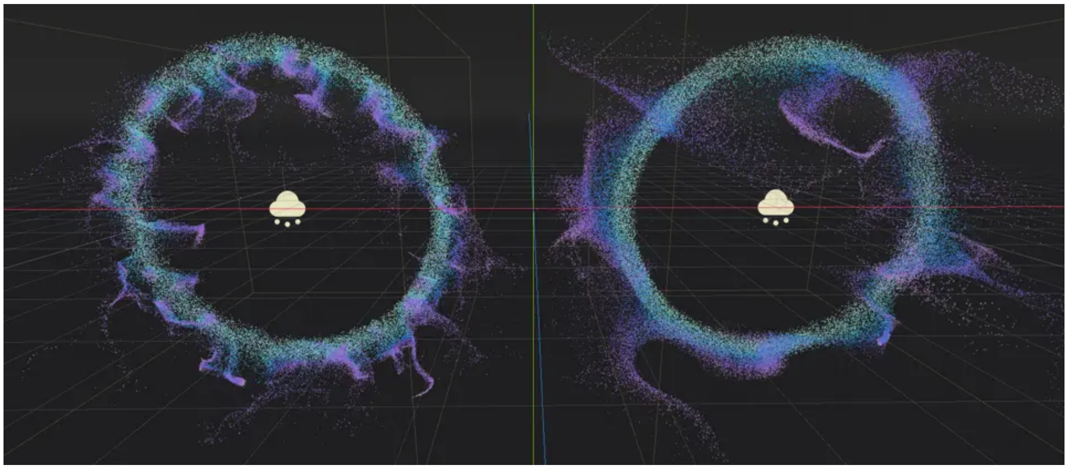
The basis for particle turbulence is a noise pattern. There are several properties that allow you to manipulate different attributes of this pattern.

The `Noise Strength` property controls the pattern's contrast, which affects the overall turbulence sharpness. A lower value creates a softer pattern where individual movement paths are not as sharply separated from another. Set this to a higher number to make the pattern more distinct.



At a value of 1 (left), the noise strength produces softer turbulence patterns than at 20 (right)

The `Noise Scale` property controls the pattern's frequency. It basically changes the noise texture's UV scale where a smaller value produces finer detail, but repeating patterns become noticeable faster. A larger value results in a weaker turbulence pattern overall, but the particle system can cover a larger area before repetition starts to become an issue.



Turbulence noise scale produces finer details at a value of 1.5 (left) than at 6 (right)

The `Noise Speed` property takes a vector and controls the noise panning speed and direction. This allows you to move the noise pattern over time, which adds another layer of movement variation to the particle system.

Warning

Don't mix up particle movement speed and noise panning speed! They are two different things. Particle movement is determined by a number of properties, including the turbulence noise. The `Noise`

`Speed` property moves the pattern itself, which in turn changes where the noise affects the particles.

At a value of `(X=0,Y=0,Z=0)`, the noise pattern doesn't move at all. The influence on particle movement stays the same at any given point. Set the speed to `(X=1,Y=0,Z=0)` instead, and the noise pattern moves along the X-axis.

Turbulence noise speed

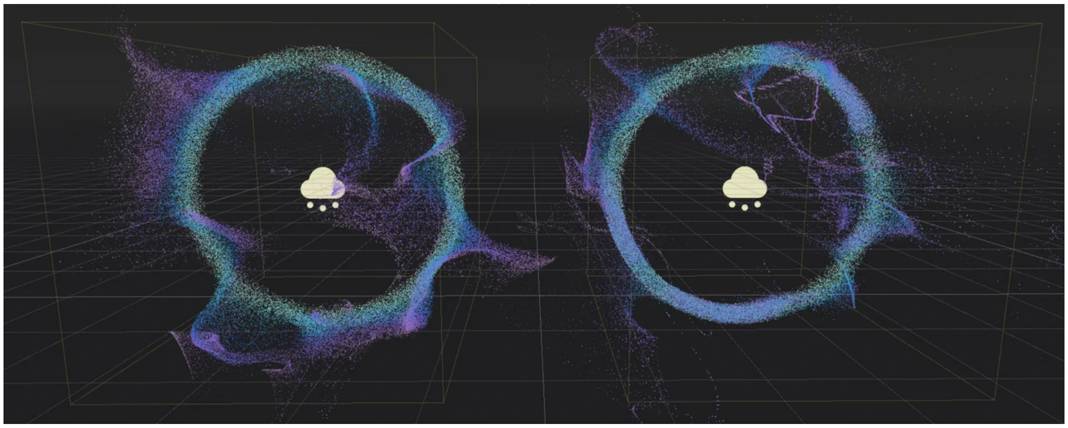
Different noise speed values. Left: (X=0,Y=0,Z=0), middle: (X=0.5,Y=0.5,Z=0.5), right: (X=0,Y=-2,Z=0).

The `Noise Speed Random` property adds some randomness to the noise panning speed. This helps with breaking up visible patterns, especially at higher panning speeds when repetition becomes noticeable faster.

Influence properties

The influence properties determine how much each particle is affected by turbulence. Use `Influence Min` to set a minimum value and `Influence Max` to set a maximum value. When a particle spawns, the influence is randomly chosen from within this range. You can also set up a curve with the `Influence Over Life` property that modifies that value over each particle's lifetime. These three properties together control the strength of the turbulence's effect on the particle system as described before.

Since these properties affect the overall influence of the turbulence over a particle system, both movement direction and speed change as you set different values. A stronger influence causes a particle to move faster and all particles to follow along narrower paths as a result of that.

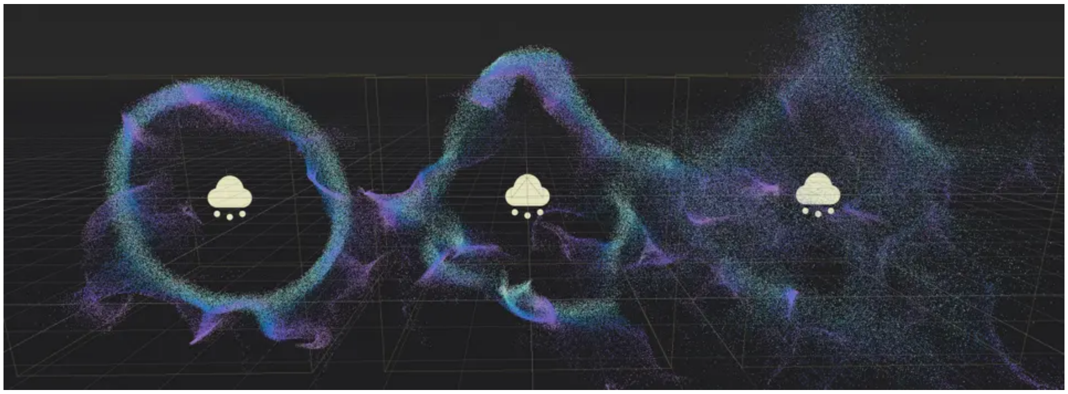


Notice how the particle paths are more narrow and less spread out at high influence values (right)

Displacement properties

Displacement changes a particle's starting position. Use `Initial Displacement Min` to set a lower limit and `Initial Displacement Max` to set an upper limit. When a particle spawns, the amount of displacement is randomly chosen from within this range and multiplied by a random direction.

Displacement is very useful to break up regular shapes or to create complex shapes from simpler ones. The only difference between the particle systems in the screenshot below is the value given to the displacement properties.



No displacement (left), displacement value of 5 (middle), displacement range [-20, 20] (right)

3D Particle attractors

Particle attractors

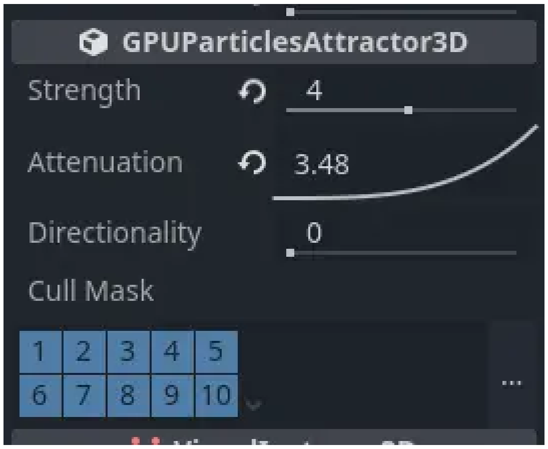
Particle attractors are nodes that apply a force to all particles within their reach. They pull particles closer or push them away based on the direction of that force. There are three types of attractors: GPUParticlesAttractorBox3D, GPUParticlesAttractorSphere3D, and GPUParticlesAttractorVectorField3D. You can instantiate them at runtime and change their properties from gameplay code; you can even animate and combine them for complex attraction effects.

Note

Particle attractors are not yet implemented for 2D particle systems.

The first thing you have to do if you want to use attractors is enable the `Attractor Interaction` property on the ParticleProcessMaterial. Do this for every particle system that needs to react to attractors. Like most properties in Godot, you can also change this at runtime.

Common properties



Strength 4

Attenuation 3.48

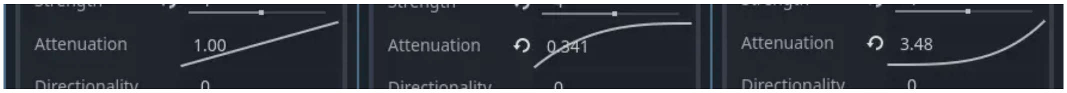
Directionality 0

Common attractor properties

There are some properties that you can find on all attractors. They're located in the `GPUParticlesAttractor3D` section in the inspector.

`Strength` controls how strong the attractor force is. A positive value pulls particles closer to the attractor's center, while a negative value pushes them away.

`Attenuation` controls the strength falloff within the attractor's influence region. Every particle attractor has a boundary. Its strength is weakest at the border of this boundary and strongest at its center. Particles outside of the boundary are not affected by the attractor at all. The attenuation curve controls how the strength weakens over that distance. A straight line means that the strength is proportional to the distance: if a particle is halfway between the boundary and the center, the attractor strength will be half of what it is at the center. Different curve shapes change how fast particles accelerate towards the attractor.



Attenuation 1.00

Attenuation 0.341

Attenuation 3.48

Strength increase variations: constantly over the distance to the attractor (left), fast at the boundary border and slowly at the center (middle), slowly at the boundary and fast at the center (right).

The `Directionality` property changes the direction towards which particles are pulled. At a value of `0.0`, there is no directionality, which means that particles are pulled towards the attractor's center. At `1.0`, the attractor is fully directional, which means particles will be pulled along the attractor's local `-Z`-axis. You can change the global direction by rotating the attractor. If `Strength` is negative, particles are instead pulled along the `+Z`-axis.

Different attractor directionality values

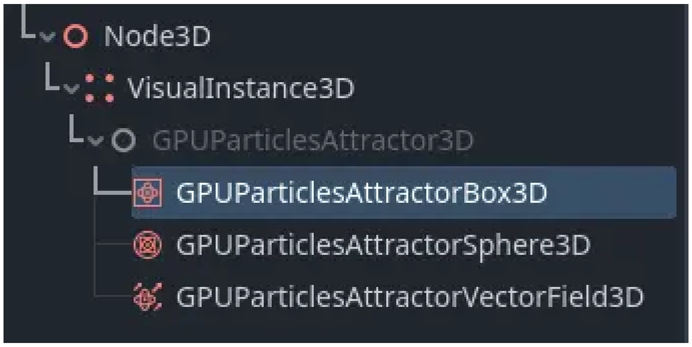
No directionality (left) vs. full directionality (right). Notice how the particles move along the attractor's local Z-axis.

The `Cull Mask` property controls which particle systems are affected by an attractor based on each system's visibility layers. A particle system is only affected by an attractor if at least one of the system's visibility layers is enabled in the attractor's cull mask.

Warning

There is a known issue [https://github.com/godotengine/godot/issues/61014] with GPU particle attractors that prevent the cull mask from working properly in Godot 4.0. We will update the documentation as soon as it is fixed.

Box attractors



Node3D
VisualInstance3D
GPUParticlesAttractor3D
GPUParticlesAttractorBox3D
GPUParticlesAttractorSphere3D
GPUParticlesAttractorVectorField3D

Box attractor in the node list

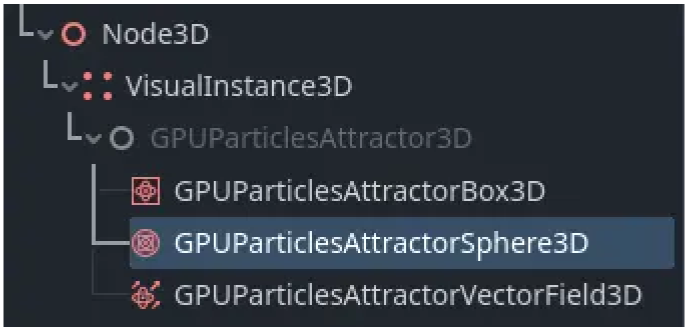
Box attractors have a box-shaped influence region. You control their size with the `Extents` property. Box extents always measure half of the sides of its bounds, so a value of `(X=1.0,Y=1.0,Z=1.0)` creates a box with an influence region that is 2 meters wide on each side.

To create a box attractor, add a new child node to your scene and select `GPUParticlesAttractorBox3D` from the list of available nodes. You can animate the box position or attach it to a moving node for more dynamic effects.

Box attractor parts particle field

A box attractor with a negative strength value parts a particle field as it moves through it.

Sphere attractors



Node3D
VisualInstance3D
GPUParticlesAttractor3D
GPUParticlesAttractorBox3D
GPUParticlesAttractorSphere3D
GPUParticlesAttractorVectorField3D

Sphere attractor in the node list

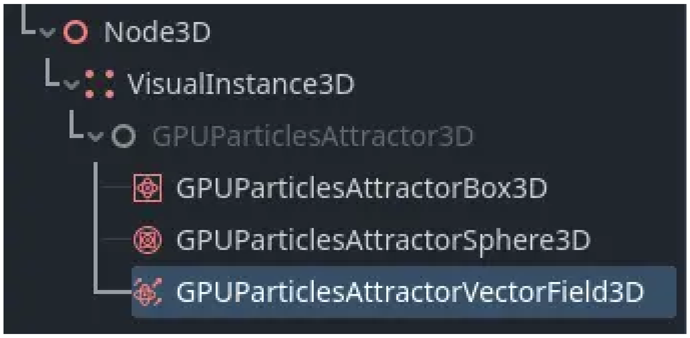
Sphere attractors have a spherical influence region. You control their size with the `Radius` property. While box attractors don't have to be perfect cubes, sphere attractors will always be spheres: You can't set width independently from height. If you want to use a sphere attractor for elongated shapes, you have to change its `Scale` in the attractor's `Node3D` section.

To create a sphere attractor, add a new child node to your scene and select `GPUParticlesAttractorSphere3D` from the list of available nodes. You can animate the sphere position or attach it to a moving node for more dynamic effects.

Sphere attractor parts particle field

A sphere attractor with a negative strength value parts a particle field as it moves through it.

Vector field attractors


Node3D
VisualInstance3D
GPUParticlesAttractor3D
GPUParticlesAttractorBox3D
GPUParticlesAttractorSphere3D
GPUParticlesAttractorVectorField3D

Vector field attractor in the node list

A vector field is a 3D area that contains vectors positioned on a grid. The grid density controls how many vectors there are and how far they're spread apart. Each vector in a vector field points in a specific direction. This can be completely random or aligned in a way that forms distinct patterns and paths.

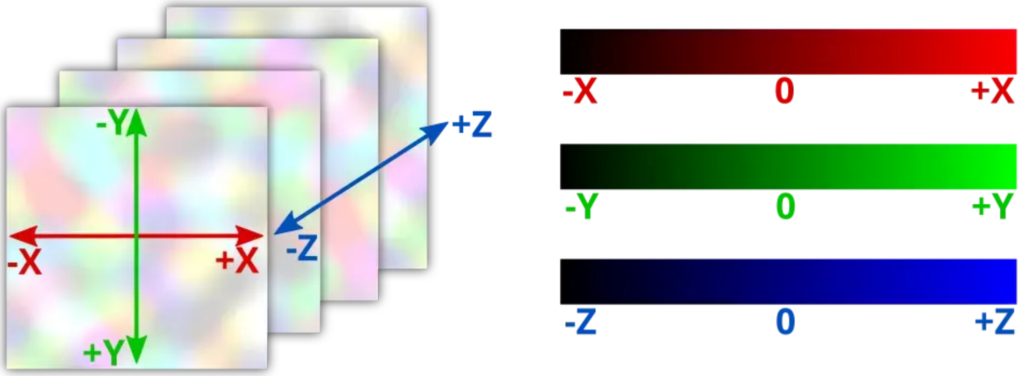
When particles interact with a vector field, their movement direction changes to match the nearest vector in the field. As a particle moves closer to the next vector in the field, it changes direction to match that vector's direction. The particle's speed depends on the vector's length.

Like box attractors, vector field attractors have a box-shaped influence region. You control their size with the `Extents` property, where a value of `(X=1.0,Y=1.0,Z=1.0)` creates a box with an influence region that is 2 meters wide on each side. The `Texture` property takes a 3D texture where every pixel represents a vector with the pixel's color interpreted as the vector's direction and size.

Note

When a texture is used as a vector field, there are two types of conversion you need to be aware of:

1. The texture coordinates map to the attractor bounds. The image below shows which part of the texture corresponds to which part of the vector field volume. For example, the bottom half of the texture affects the top half of the vector field attractor because `+Y` points down in the texture UV space, but up in Godot's world space.
2. The pixel color values map to direction vectors in space. The image below provides an overview. Since particles can move in two directions along each axis, the lower half of the color range represents negative direction values while the upper half represents positive direction values. So a yellow pixel `(R=1,G=1,B=0)` maps to the vector `(X=1,Y=1,Z=-1)` while a neutral gray `(R=0.5,G=0.5,B=0.5)` results in no movement at all.



To create a vector field attractor, add a new child node to your scene and select `GPUParticlesAttractorVectorField3D` from the list of available nodes. You can animate the attractor's position or attach it to a moving node for more dynamic effects.

Tip

If you don't have external tools to create vector field textures, you can use a NoiseTexture3D with a Color Ramp attached as a vector field texture. The Color Ramp can be modified to adjust how much each coordinate is affected by the vector field.

Vector field attractor in a field of particles

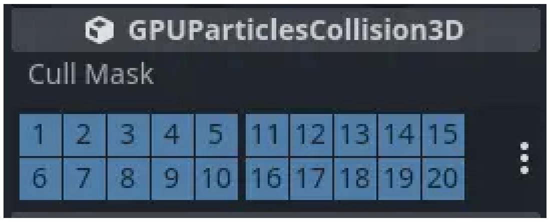
Two particle systems are affected by the same vector field attractor. **`Click here to download the 3D texture.`**

3D Particle collisions

Particle collisions

Since GPU particles are processed entirely on the GPU, they don't have access to the game's physical world. If you need particles to collide with the environment, you have to set up particle collision nodes. There are four of them: GPUParticlesCollisionBox3D, GPUParticlesCollisionSphere3D, GPUParticlesCollisionSDF3D, and GPUParticlesCollisionHeightField3D.

Common properties



Common collision properties

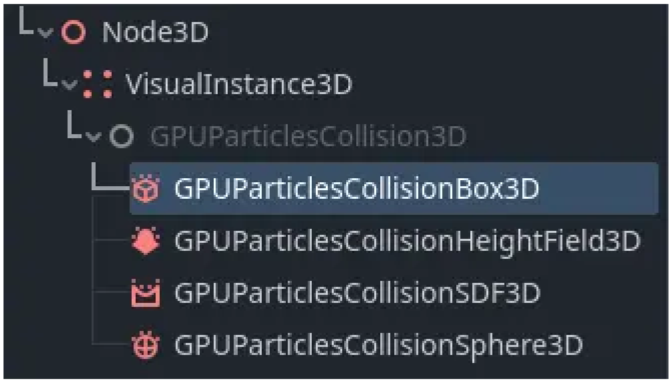
There are some properties that you can find on all collision nodes. They're located in the `GPUParticlesCollision3D` section in the inspector.

The `Cull Mask` property controls which particle systems are affected by a collision node based on each system's visibility layers. A particle system collides with a collision node only if at least one of the system's visibility layers is enabled in the collider's cull mask.

Warning

There is a known issue [https://github.com/godotengine/godot/issues/61014] with GPU particle collision that prevent the cull mask from working properly in Godot 4.0. We will update the documentation as soon as it is fixed.

Box collision


Node3D
VisualInstance3D
GPUParticlesCollision3D
GPUParticlesCollisionBox3D
GPUParticlesCollisionHeightField3D
GPUParticlesCollisionSDF3D
GPUParticlesCollisionSphere3D

Box collision in the node list

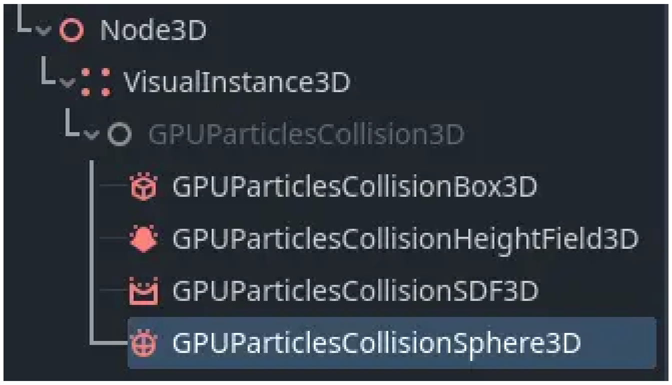
Box collision nodes are shaped like a solid, rectangular box. You control their size with the `Extents` property. Box extents always measure half of the sides of its bounds, so a value of `(X=1.0,Y=1.0,Z=1.0)` creates a box that is 2 meters wide on each side. Box collision nodes are useful for simulating floor and wall geometry that particles should collide against.

To create a box collision node, add a new child node to your scene and select `GPUParticlesCollisionBox3D` from the list of available nodes. You can animate the box position or attach it to a moving node for more dynamic effects.

Box collision with particle systems

Two particle systems collide with a box collision node

Sphere collision


Node3D
VisualInstance3D
GPUParticlesCollision3D
GPUParticlesCollisionBox3D
GPUParticlesCollisionHeightField3D
GPUParticlesCollisionSDF3D
GPUParticlesCollisionSphere3D

Sphere collision in the node list

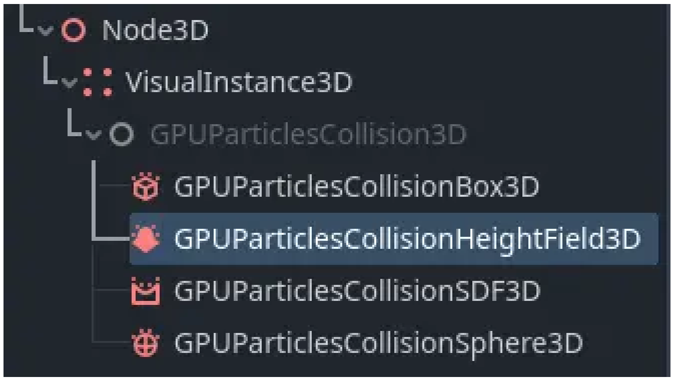
Sphere collision nodes are shaped like a solid sphere. The `Radius` property controls the size of the sphere. While box collision nodes don't have to be perfect cubes, sphere collision nodes will always be spheres. If you want to set width independently from height, you have to change the `Scale` property in the `Node3D` section.

To create a sphere collision node, add a new child node to your scene and select `GPUParticlesCollisionSphere3D` from the list of available nodes. You can animate the sphere's position or attach it to a moving node for more dynamic effects.

Sphere collision with particle systems

Two particle systems collide with a sphere collision node

Height field collision



Node3D
VisualInstance3D
GPUParticlesCollision3D
GPUParticlesCollisionBox3D
GPUParticlesCollisionHeightField3D
GPUParticlesCollisionSDF3D
GPUParticlesCollisionSphere3D

Height field collision in the node list

Height field particle collision is very useful for large outdoor areas that need to collide with particles. At runtime, the node creates a height field from all the meshes within its bounds that match its cull mask. Particles collide against the mesh that this height field represents. Since the height field generation is done dynamically, it can follow the player camera around and react to changes in the level. Different settings for the height field density offer a wide range of performance adjustments.

To create a height field collision node, add a new child node to your scene and select `GPUParticlesCollisionHeightField3D` from the list of available nodes.

A height field collision node is shaped like a box. The `Extents` property controls its size. Extents always measure half of the sides of its bounds, so a value of `(X=1.0,Y=1.0,Z=1.0)` creates a box that is 2 meters wide on each side. Anything outside of the node's extents is ignored for height field creation.

The `Resolution` property controls how detailed the height field is. A lower resolution performs faster at the cost of accuracy. If the height field resolution is too low, it may look like particles penetrate level geometry or get stuck in the air during collision events. They might also ignore some smaller meshes completely.

Height field resolutions

At low resolutions, height field collision misses some finer details (left)

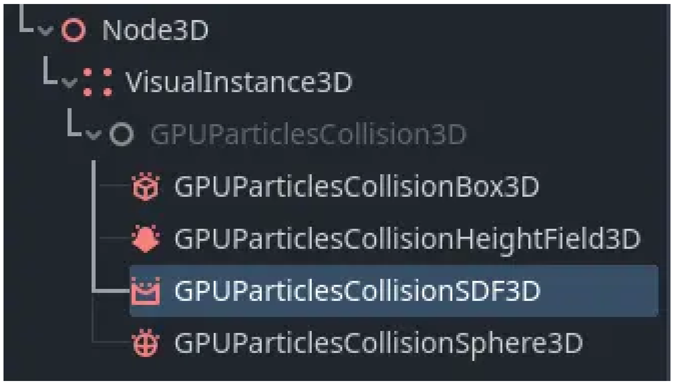
The `Update Mode` property controls when the height field is recreated from the meshes within its bounds. Set it to `When Moved` to make it refresh only when it moves. This performs well and is suited for static scenes that don't change very often. If you need particles to collide with dynamic objects that change position frequently, you can select `Always` to refresh every frame. This comes with a cost to performance and should only be used when necessary.

Note

It's important to remember that when `Update Mode` is set to `When Moved`, it is the *height field node* whose movement triggers an update. The height field is not updated when one of the meshes inside it moves.

The `Follow Camera Enabled` property makes the height field follow the current camera when enabled. It will update whenever the camera moves. This property can be used to make sure that there is always particle collision around the player while not wasting performance on regions that are out of sight or too far away.

SDF collision


Node3D
VisualInstance3D
GPUParticlesCollision3D
GPUParticlesCollisionBox3D
GPUParticlesCollisionHeightField3D
GPUParticlesCollisionSDF3D
GPUParticlesCollisionSphere3D

SDF collision in the node list

SDF collision nodes create a signed distance field [https://www.reddit.com/r/explainlikeimfive/comments/k2zbos/eli5_what_are_distance_fields_in_graphics] that particles can collide with. SDF collision is similar to height field collision in that it turns multiple meshes within its bounds into a single collision volume for particles. A major difference is that signed distance fields can represent holes, tunnels and overhangs, which is impossible to do with height fields alone. The performance overhead is larger compared to height fields, so they're best suited for small-to-medium-sized environments.

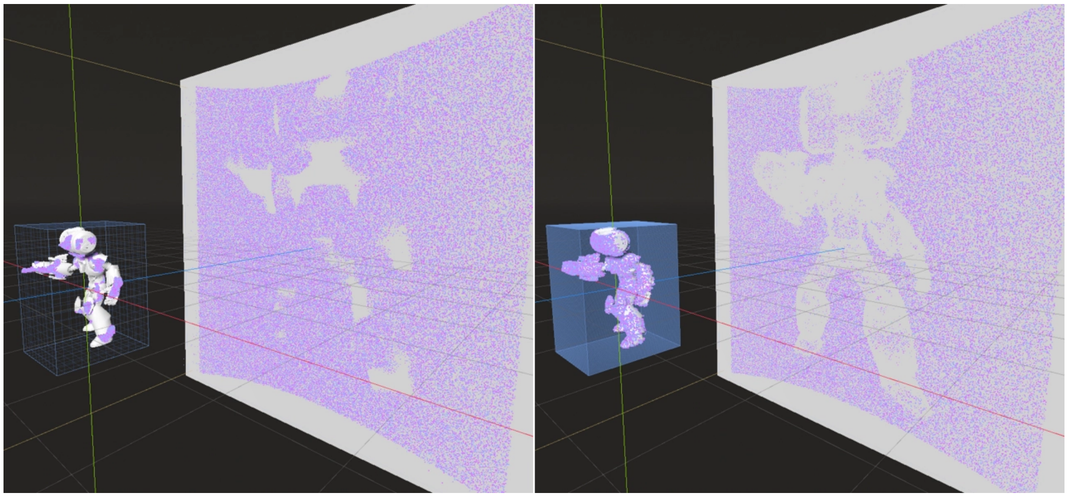
To create an SDF collision node, add a new child node to your scene and select `GPUParticlesCollisionSDF3D` from the list of available nodes. SDF collision nodes have to be baked in order to have any effect on particles in the level. To do that, click the `Bake SDF` button in the viewport toolbar while the SDF collision node is selected and choose a directory to store the baked data. Since SDF collision needs to be baked in the editor, it's static and cannot change at runtime.

SDF particle collision

SDF particle collision allows for very detailed 3-dimensional collision shapes

An SDF collision node is shaped like a box. The `Extents` property controls its size. Extents always measure half of the sides of its bounds, so a value of `(X=1.0,Y=1.0,Z=1.0)` creates a box that is 2 meters wide on each side. Anything outside of the node's extents is ignored for collision.

The `Resolution` property controls how detailed the distance field is. A lower resolution performs faster at the cost of accuracy. If the resolution is too low, it may look like particles penetrate level geometry or get stuck in the air during collision events. They might also ignore some smaller meshes completely.



The same area covered by a signed distance field at different resolutions: 16 (left) and 256 (right)

The `Thickness` property gives the distance field, which is usually hollow on the inside, a thickness to prevent particles from penetrating at high speeds. If you find that some particles don't collide with the level geometry and instead shoot right through it, try setting this property to a higher value.

The `Bake Mask` property controls which meshes will be considered when the SDF is baked. Only meshes that render on the active layers in the bake mask contribute to particle collision.

Troubleshooting

For particle collision to work, the particle's visibility AABB must overlap with the collider's AABB. If collisions appear to be not working despite colliders being set up, generate an updated visibility AABB by selecting the GPUParticles3D node and choosing **GPUParticles3D > Generate Visibility AABB...** at the top of the 3D editor viewport.

If the particles move fast and colliders are thin. There are two solutions for this:

- Make the colliders thicker. For instance, if particles cannot get below a solid floor, you could make the collider representing the floor thicker than its actual visual representation. The heightfield collider automatically handles this by design, as heightfields cannot represent "room over room" collision.
- Increased `Fixed FPS` in the GPUParticles3D node, which will perform collision checks more often. This comes at a performance cost, so avoid setting this too high.

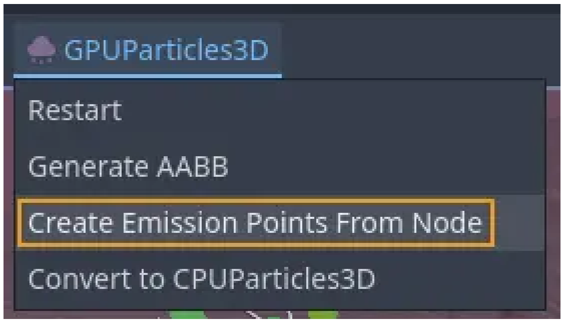
Complex emission shapes

Complex emission shapes

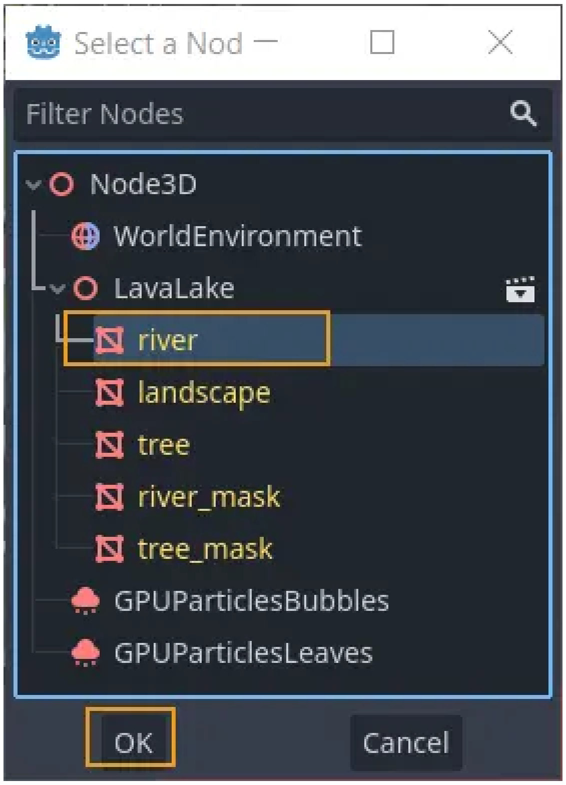
When it is not enough to emit particles from one of the simple shapes available in the process material, Godot provides a way to emit particles from arbitrary, complex shapes. The shapes are generated from meshes in the scene and stored as textures in the particle process material. This is a very versatile workflow that has allowed users to use particle systems for things that go beyond traditional use cases, like foliage, leaves on a tree, or complex holographic effects.

Note

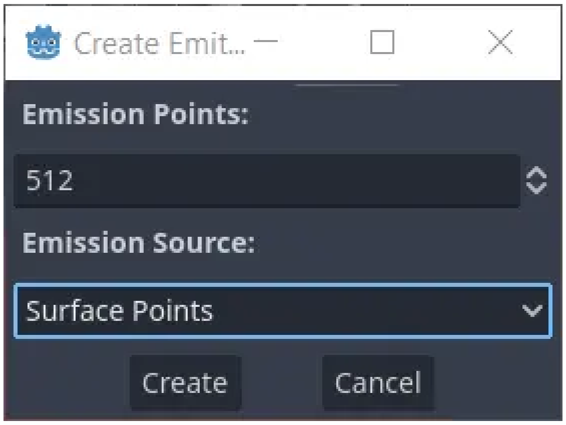
When you create emission points from meshes, you can only select a single node as emission source. If you want particles to emit from multiple shapes, you either have to create several particle systems or combine the meshes into one in an external DCC software.


GPUParticles3D
Restart
Generate AABB
Create Emission Points From Node
Convert to CPUParticles3D

Create particle emission points...


Select a Nod
Filter Nodes
Node3D
WorldEnvironment
LavaLake
river
landscape
tree
river_mask
tree_mask
GPUParticlesBubbles
GPUParticlesLeaves
OK
Cancel

...from a mesh instance as the source


Create Emit...
Create
Cancel

Emission Points: 512

Emission Source: Surface Points

More points = higher particle density

To make use of this feature, start by creating a particle system in the current scene. Add a mesh instance that serves as the source of the particle emission points. With the particle system selected, navigate to the viewport menu and select the *GPUParticles3D* entry. From there, select `Create Emission Points From Node`.

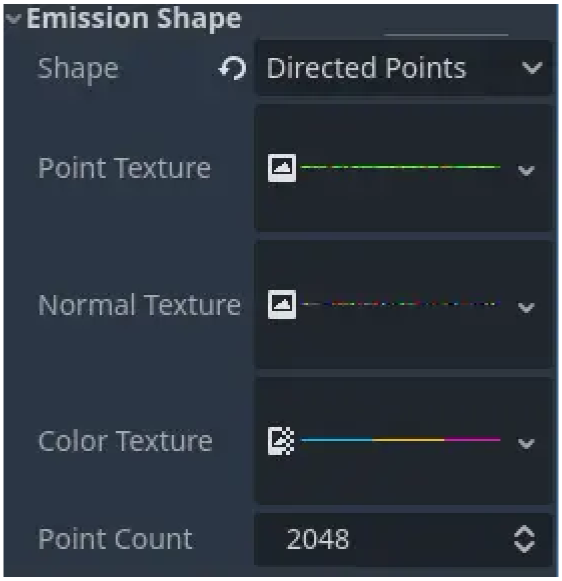
A dialog window will pop up and ask you to select a node as the emission source. Choose one of the mesh instances in the scene and confirm your selection. The next dialog window deals with the amount of points and how to generate them.

`Emission Points` controls the total number of points that you are about to generate. Particles will spawn from these points, so what to enter here depends on the size of the source mesh (how much area you have to cover) and the desired density of the particles.

`Emission Source` offers 3 different options for how the points are generated. Select `Surface Points` if all you want to do is distribute the emission points across the surface of the mesh. Select `Surface Points + Normal (Directed)` if you also want to generate information about the surface normals and make particles move in the direction that the normals point at. The last option, `Volume`, creates emission points everywhere inside the mesh, not just across its surface.

The emission points are stored in the particle system's local coordinate system, so you can move the particle node around and the emission points will follow. This might be useful when you want to use the same particle system in several different places. On the other hand, you might have to regenerate the emission points when you move either the particle system or the source mesh.

Emission shape textures



Emission Shape

Shape: Directed Points

Point Texture

Normal Texture

Color Texture

Point Count: 2048

The available emission shape textures

All the data for complex particle emission shapes is stored in a set of textures. How many, depends on the type of emission shape you use. If you set the `Shape` property in the `Emission Shape` group on the particle process material to `Points`, you have access to 2 texture properties, the `Point Texture` and the `Color Texture`. Set it to `Directed Points` and there is a third property called `Normal Texture`.

`Point Texture` contains all possible emission points that were generated in the previous step. A point is randomly selected for every particle when it spawns. `Normal Texture`, if it exists, provides a direction vector at that same location. If the `Color Texture` property is also set, it provides color for the particle, sampled at the same location as the other two textures and modulating any other color that was set up on the process material.

There is also the `Point Count` property that you can use to change the number of emission points at any time after creating the emission shape. This includes dynamically at runtime while the playing the game.

High dynamic range lighting

Introduction

Normally, an artist does all the 3D modeling, then all the texturing, looks at their awesome looking model in the 3D modeling software and says "looks fantastic, ready for integration!" then goes into the game, lighting is setup and the game runs.

So at what point does all this "HDR" business come into play? To understand the answer, we need to look at how displays behave.

Your display outputs linear light ratios from some maximum to some minimum intensity. Modern game engines perform complex math on linear light values in their respective scenes. So what's the problem?

The display has a limited range of intensity, depending on the display type. The game engine renders to an unlimited range of intensity values, however. While "maximum intensity" means something to an sRGB display, it has no bearing in the game engine; there is only a potentially infinitely wide range of intensity values generated per frame of rendering.

This means that some transformation of the scene light intensity, also known as *scene-referred* light ratios, need to be transformed and mapped to fit within the particular output range of the chosen display. This can be most easily understood if we consider virtually photographing our game engine scene through a virtual camera. Here, our virtual camera would apply a particular camera rendering transform to the scene data, and the output would be ready for display on a particular display type.

Note

Godot does not support high dynamic range *output* yet. It can only perform lighting in HDR and tonemap the result to a low dynamic range image.

For advanced users, it is still possible to get a non-tonemapped image of the viewport with full HDR data, which can then be saved to an OpenEXR file.

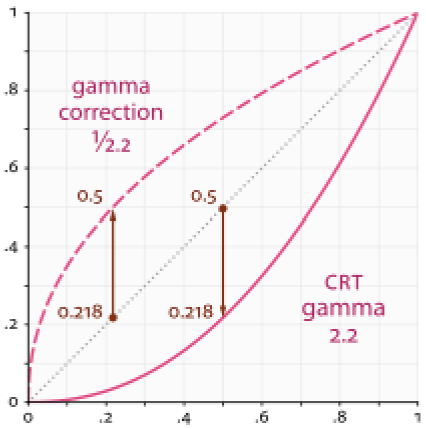
Computer displays

Almost all displays require a nonlinear encoding for the code values sent to them. The display in turn, using its unique transfer characteristic, "decodes" the code value into linear light ratios of output, and projects the ratios out of the uniquely colored lights at each reddish, greenish, and blueish emission site.

For a majority of computer displays, the specifications of the display are outlined in accordance with IEC 61966-2-1, also known as the 1996 sRGB specification. This specification outlines how an sRGB display is to behave, including the color of the lights in the LED pixels as well as the transfer characteristics of the input (OETF) and output (EOTF).

Not all displays use the same OETF and EOTF as a computer display. For example, television broadcast displays use the BT.1886 EOTF. However, Godot currently only supports sRGB displays.

The sRGB standard is based around the nonlinear relationship between the current to light output of common desktop computing CRT displays.



The mathematics of a scene-referred model require that we multiply the scene by different values to adjust the intensities and exposure to different light ranges. The transfer function of the display can't appropriately render the wider dynamic range of the game engine's scene output using the simple transfer function of the display. A more complex approach to encoding is required.

Scene linear & asset pipelines

Working in scene-linear sRGB is more complex than pressing a single switch. First, imported image assets must be converted to linear light ratios on import. Even when linearized, those assets may not be perfectly well-suited for use as textures, depending on how they were generated.

There are two ways to do this:

sRGB transfer function to display linear ratios on image import

This is the easiest method of using sRGB assets, but it's not the most ideal. One issue with this is loss of quality. Using 8 bits per channel to represent linear light ratios is not sufficient to quantize the values correctly. These textures may also be compressed later, which can exacerbate the problem.

Hardware sRGB transfer function to display linear conversion

The GPU will do the conversion after reading the texel using floating-point. This works fine on PC and consoles, but most mobile devices don't support it, or they don't support it on compressed texture formats (iOS for example).

Scene linear to display-referred nonlinear

After all the rendering is done, the scene linear render requires transforming to a suitable output such as an sRGB display. To do this, enable sRGB conversion in the current Environment (more on that below).

Keep in mind that the **sRGB -> Display Linear** and **Display Linear -> sRGB** conversions must always be **both** enabled. Failing to enable one of them will result in horrible visuals suitable only for avant-garde experimental indie games.

Parameters of HDR

HDR settings can be found in the Environment resource. Most of the time, these are found inside a WorldEnvironment node or set in a Camera node. For more information, see Environment and post-processing.

Global illumination

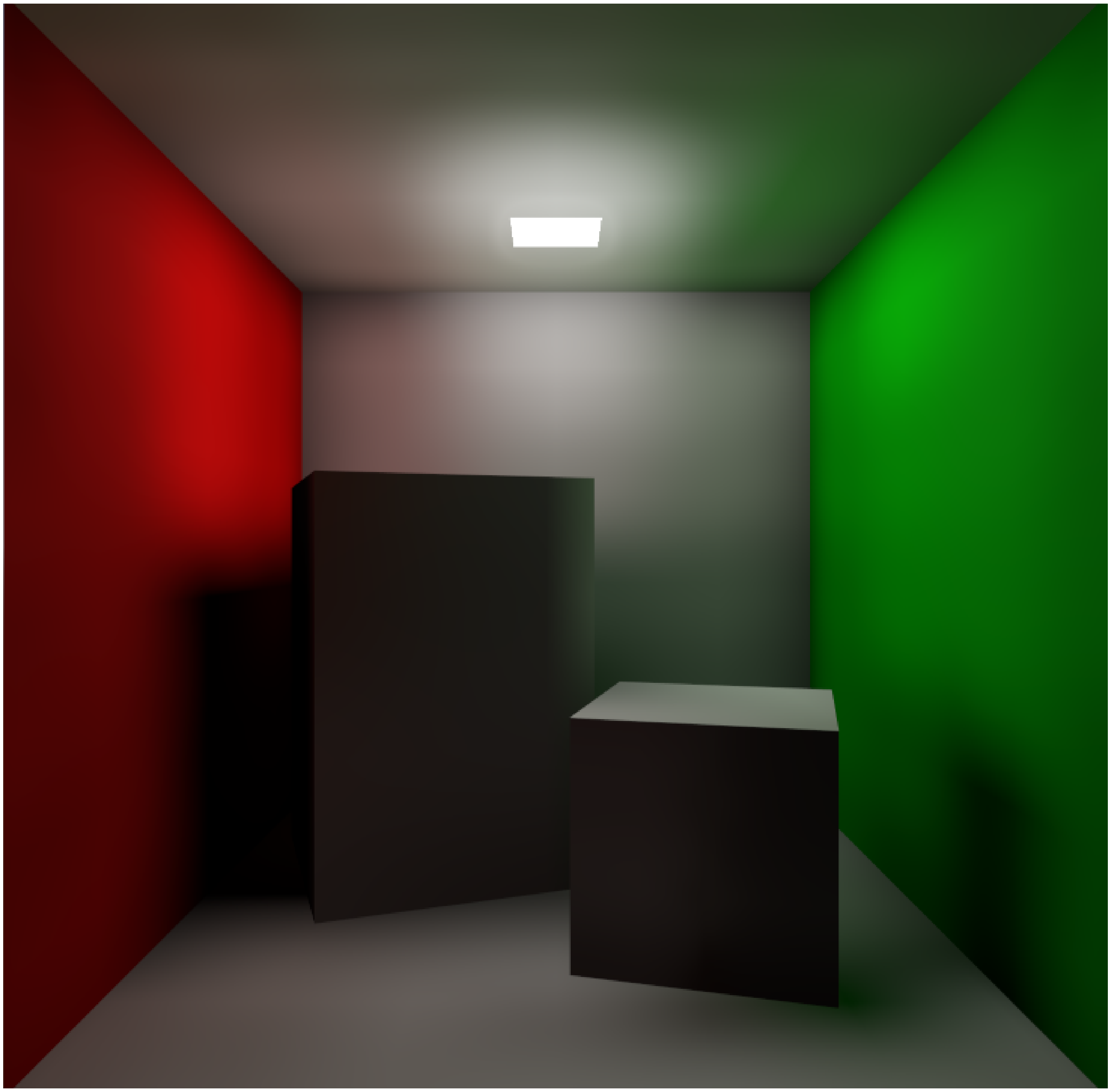
- Introduction to global illumination
- Using Voxel global illumination
- Signed distance field global illumination (SDFGI)
- Using Lightmap global illumination
- Reflection probes
- Faking global illumination

Introduction to global illumination

What is global illumination?

Global illumination is a catch-all term used to describe a system of lighting that uses both direct light (light that comes directly from a light source) and indirect light (light that bounces from a surface). In a 3D rendering engine, global illumination is one of the most important elements to achieving realistic lighting. Global illumination aims to mimic how light behaves in real life, such as light bouncing on surfaces and light being emitted from emissive materials.

In the example below, the entire scene is illuminated by an emissive material (the white square at the top). The white wall and ceiling on the back is tinted red and green close to the walls, as the light bouncing on the colored walls is being reflected back onto the rest of the scene.



Global illumination is composed of several key concepts:

Indirect diffuse lighting

This is the lighting that does not change depending on the camera's angle. There are two main sources of indirect diffuse lighting:

- Light *bouncing* on surfaces. This bounced lighting is multiplied with the material's albedo color. The bounced lighting can then

be reflected by other surfaces, with decreasing impact due to light attenuation. In real life, light bounces an infinite number of times. However, for performance reasons, this can't be simulated in a game engine. Instead, the number of bounces is typically limited to 1 or 2 (or up to 16 when baking lightmaps). A greater number of bounces will lead to more realistic light falloff in shaded areas, at the cost of lower performance or greater bake times.
- Emissive materials can also emit light that can be bounced on surfaces. This acts as a form of *area lighting*. Instead of having an infinitely small point emit light using an OmniLight3D or SpotLight3D node, an area of a determined size will emit light using its own surface.

Direct diffuse lighting is already handled by the light nodes themselves, which means that global illumination algorithms only try to represent indirect lighting.

Different global illumination techniques offer varying levels of accuracy to represent indirect diffuse lighting. See the comparison table at the bottom of this page for more information.

To provide more accurate ambient occlusion for small objects, screen-space ambient occlusion (SSAO) can be enabled in the environment settings. SSAO has a significant performance cost, so make sure to disable it when targeting low-end hardware.

Note

Indirect diffuse lighting may be a source of color banding in scenes with no detailed textures. This results in light gradients not being smooth, but having a visible "stepping" effect instead. See the Color banding section in the 3D rendering limitations documentation for ways to reduce this effect.

Specular lighting

Specular lighting is also referred to as *reflections*. This is the lighting that changes in intensity depending on the camera's angle. This specular lighting can be *direct* or *indirect*.

Most global illumination techniques offer a way to render specular lighting. However, the degree of accuracy at which specular lighting is rendered varies greatly from technique to technique. See the comparison table at the bottom of this page for more information.

To provide more accurate reflections for small objects, screen-space reflections (SSR) can be enabled in the environment settings. SSR has a significant performance cost (even more so than SSAO), so make sure to disable it when targeting low-end hardware.

Which global illumination technique should I use?

When determining a global illumination (GI) technique to use, there are several criteria to keep in mind:

- **Performance.** Real-time GI techniques are usually more expensive compared to semi-real-time or baked techniques. Note that most of the cost in GI rendering is spent on the GPU, rather than the CPU.
- **Visuals.** On top of not performing the best, real-time GI techniques generally don't provide the best visual output. This is especially the case in a mostly static scene where the dynamic nature of real-time GI is not easily noticeable. If maximizing visual quality is your goal, baked techniques will often look better and will result in fewer light leaks.
- **Real-time ability.** Some GI techniques are fully real-time, whereas others are only semi-real-time or aren't real-time at all.

Semi-real-time techniques have restrictions that fully real-time techniques don't. For instance, dynamic objects may not contribute emissive lighting to the scene. Non-real-time techniques do not support *any* form of dynamic GI, so it must be faked using other techniques if needed (such as placing positional lights near emissive surfaces). Real-time ability also affects the GI technique's viability in procedurally generated levels.

- **User work needed.** Some GI techniques are fully automatic, whereas others require careful planning and manual work on the user's side. Depending on your time budget, some GI techniques may be preferable to others.

Here's a comparison of all the global illumination techniques available in Godot:

Performance

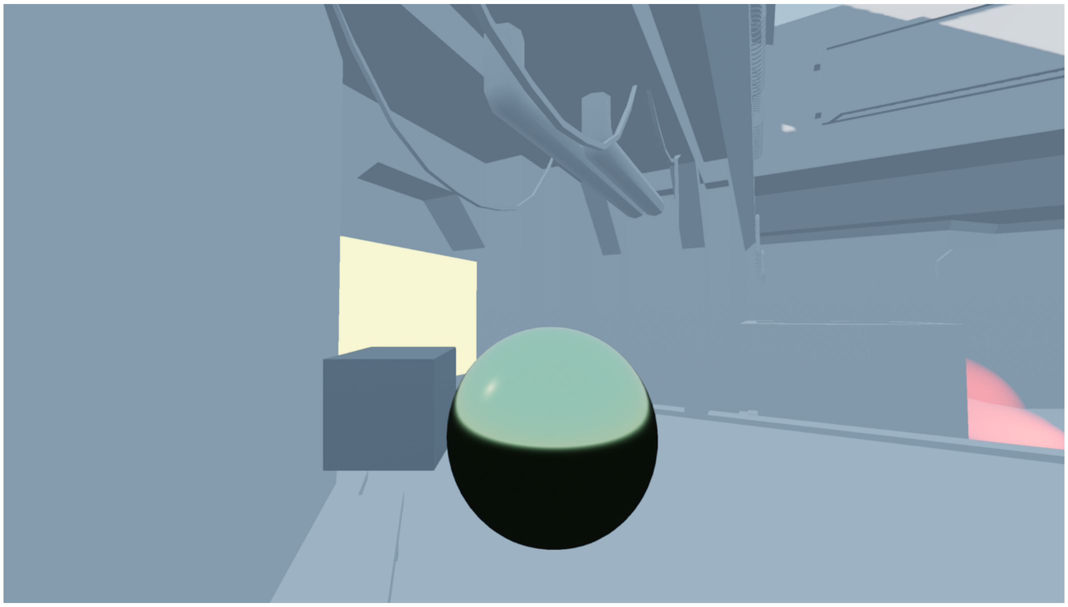
In order of performance from fastest to slowest:

- **ReflectionProbe:**
  - ReflectionProbes with their update mode set to **Always** are much more expensive than probes with their update mode set to **Once** (the default). Suited for integrated graphics when using the **Once** update mode. *Available when using the Forward Mobile backend. Will be available in the Compatibility backend in later releases.*
- **LightmapGI:**
  - Lights can be baked with indirect lighting only, or fully baked on a per-light basis to further improve performance. Hybrid setups can be used (such as having a real-time directional light and fully baked positional lights). Directional information can be enabled before baking to improve visuals at a small performance cost (and at the cost of larger file sizes). Suited for integrated graphics.

    Available when using the Forward Mobile backend. Will be available in the Compatibility backend in later releases.
- **VoxelGI:**
    - The bake's number of subdivisions can be adjusted to balance between performance and quality. The VoxelGI rendering quality can be adjusted in the Project Settings. The rendering can optionally be performed at half resolution (and then linearly scaled) to improve performance significantly. **Not available** *when using the Forward Mobile or Compatibility backends.*
- **Screen-space indirect lighting (SSIL):**
    - The SSIL quality and number of blur passes can be adjusted in the Project Settings. By default, SSIL rendering is performed at half resolution (and then linearly scaled) to ensure a reasonable performance level. **Not available** *when using the Forward Mobile or Compatibility backends.*
- **SDFGI:**
    - The number of cascades can be adjusted to balance performance and quality. The number of rays thrown per frame can be adjusted in the Project Settings. The rendering can optionally be performed at half resolution (and then linearly scaled) to improve performance significantly. **Not available** *when using the Forward Mobile or Compatibility backends.*

Visuals

For comparison, here's a 3D scene with no global illumination options used:

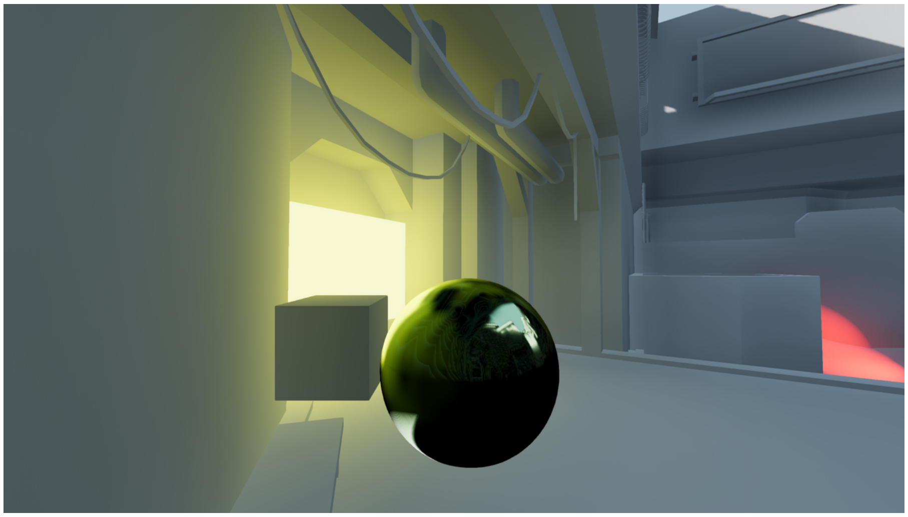


A 3D scene without any form of global illumination (only constant environment lighting). The box and sphere near the camera are both dynamic objects.

Here's how Godot's various global illumination techniques compare:

- **VoxelGI:** 🟡 Good reflections and indirect lighting, but beware of leaks.
    - Due to its voxel-based nature, VoxelGI will exhibit light leaks if walls and floors are too thin. It's recommended to make sure all solid surfaces are at least as thick as one voxel.

      Streaking artifacts may also be visible on sloped surfaces. In this case, tweaking the bias properties or rotating the VoxelGI node can help combat this.

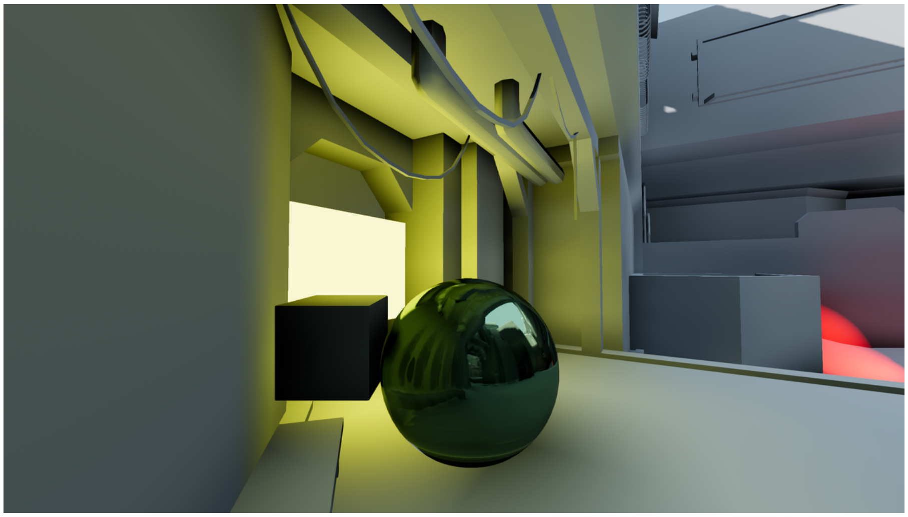


VoxelGI in action.

- **SDFGI:** ● Good reflections and indirect lighting, but beware of leaks and visible cascade shifts.
    - GI level of detail varies depending on the distance between the camera and surface.

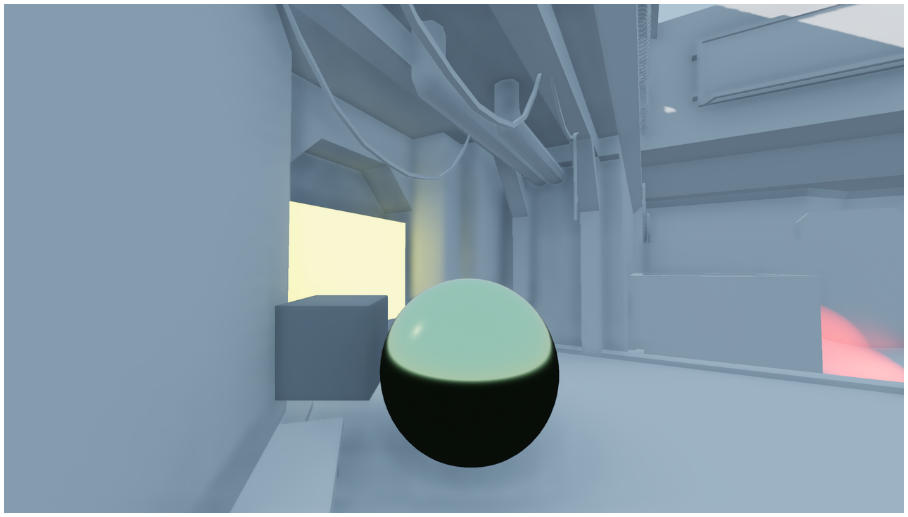
      Leaks can be reduced significantly by enabling the **Use Occlusion** property. This has a small performance cost, but it often results in fewer leaks compared to VoxelGI.

      Cascade shifts may be visible when the camera moves fast. This can be made less noticeable by adjusting the cascade sizes or using fog.



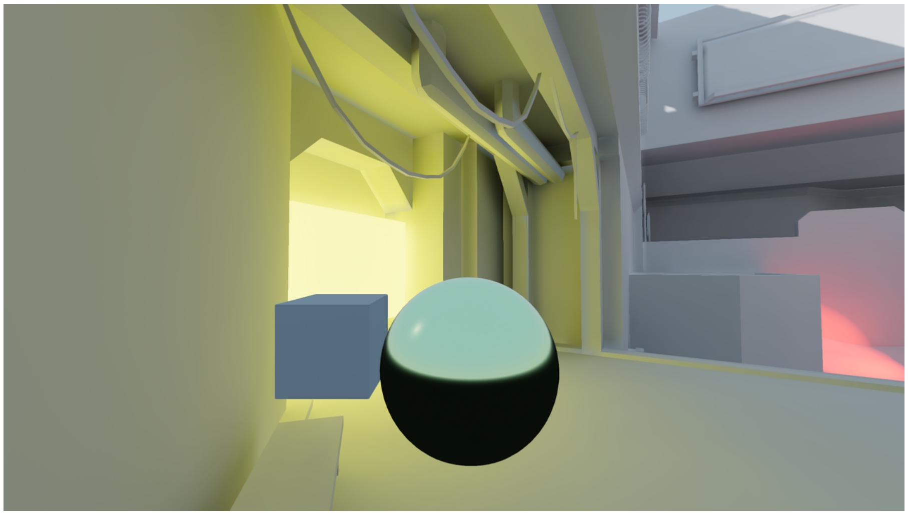
SDFGI in action.

- **Screen-space indirect lighting (SSIL):** ● Good *secondary* source of indirect lighting, but no reflections.
    - SSIL is designed to be used as a complement to another GI technique such as VoxelGI, SDFGI or LightmapGI. SSIL works best for small-scale details, as it cannot provide accurate indirect lighting for large structures on its own. SSIL can provide real-time indirect lighting in situations where other GI techniques fail to capture small-scale details or dynamic objects. Its screen-space nature will result in some artifacts, especially when objects enter and leave the screen. SSIL works using the last frame's color (before post-processing) which means that emissive decals and custom shaders are included (as long as they're present on screen).



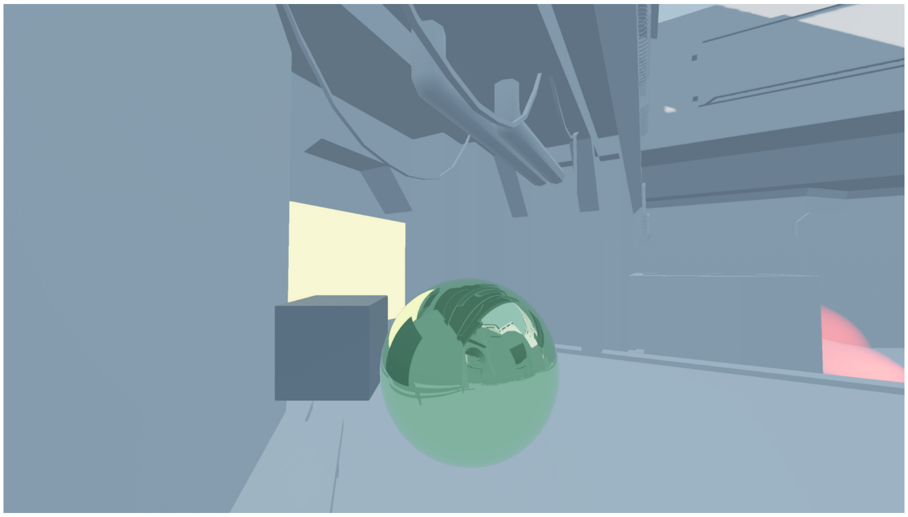
SSIL in action (without any other GI technique). Notice the emissive lighting around the yellow box.

- **LightmapGI:** 🟢 Excellent indirect lighting, decent reflections (optional).
    - This is the only technique where the number of light bounces can be pushed above 2 (up to 16). When directional information is enabled, spherical harmonics (SH) are used to provide blurry reflections.



LightmapGI in action. Only indirect lighting is baked here, but direct light can also be baked.

- **ReflectionProbe:** 🟡 Good reflections, but poor indirect lighting.
    - Indirect lighting can be disabled, set to a constant color spread throughout the probe, or automatically read from the probe's environment (and applied as a cubemap). This essentially acts as local ambient lighting. Reflections and indirect lighting are blended with other nearby probes.



ReflectionProbe in action (without any other GI technique). Notice the reflective sphere.

Real-time ability

- **VoxelGI:** 🟢 Fully real-time.
    - Indirect lighting and reflections are fully real-time. Dynamic objects can receive GI *and* contribute to it with their emissive surfaces. Custom shaders can also emit their own light, which will be emitted accurately.

      Viable for procedurally generated levels *if they are generated in advance* (and not during gameplay). Baking requires several seconds or more to complete, but it can be done from both the editor and an exported project.

- **SDFGI:** 🟡 Semi-real-time.
  - Cascades are generated in real-time, making SDFGI viable for procedurally generated levels (including when structures are generated during gameplay).

    Dynamic objects can *receive* GI, but not *contribute* to it. Emissive lighting will only update when an object enters a cascade, so it may still work for slow-moving objects.
- **Screen-space indirect lighting (SSIL):** 🟢 Fully real-time.
  - SSIL works with both static and dynamic lights. It also works with both static and dynamic occluders (including emissive materials).
- **LightmapGI:** 🔴 Baked, and therefore not real-time.
  - Both indirect lighting and SH reflections are baked and can't be changed at run-time. Real-time GI must be simulated via other means, such as real-time positional lights. Dynamic objects receive indirect lighting via light probes, which can be placed automatically or manually by the user (LightmapProbe node). Not viable for procedurally generated levels, as baking lightmaps is only possible from the editor.
- **ReflectionProbe:** 🟡 Optionally real-time.
  - By default, reflections update when the probe is moved. They update as often as possible if the update mode is set to **Always** (which is expensive).
  - Indirect lighting must be configured manually by the user, but can be changed at run-time without causing an expensive computation to happen behind the scenes. This makes ReflectionProbes viable for procedurally generated levels.

User work needed

- **VoxelGI:** One or more VoxelGI nodes need to be created and baked.
    - Adjusting extents correctly is required to get good results. Additionally rotating the node and baking again can help combat leaks or streaking artifacts in certain situations. Bake times are fast – usually below 10 seconds for a scene of medium complexity.
- **SDFGI:** Very little.
    - SDFGI is fully automatic; it only needs to be enabled in the Environment resource. The only manual work required is to set MeshInstances' bake mode property correctly. No node needs to be created, and no baking is required.
- **Screen-space indirect lighting (SSIL):** Very little.
    - SSIL is fully automatic; it only needs to be enabled in the Environment resource. No node needs to be created, and no baking is required.
- **LightmapGI:** Requires UV2 setup and baking.
    - Static meshes must be reimported with UV2 and lightmap generation enabled. On a dedicated GPU, bake times are relatively fast thanks to the GPU-based lightmap baking – usually below 1 minute for a scene of medium complexity.
- **ReflectionProbe:** Placed manually by the user.

Summary

If you are unsure about which GI technique to use:

- For desktop games, it's a good idea to start with SDFGI first as it requires the least amount of setup. Move to other GI techniques later if needed. To improve performance on low-end GPUs and integrated graphics, consider adding an option to disable SDFGI or VoxelGI in your game's settings. SDFGI can be disabled in the Environment resource, and VoxelGI can be

disabled by hiding the VoxelGI node(s). To further improve visuals on high-end setups, add an option to enable SSIL in your game's settings.
- For mobile games, LightmapGI and ReflectionProbes are the only supported options. See also Alternatives to GI techniques.

See also

You can compare global illumination techniques in action using the Global Illumination demo project [https://github.com/godotengine/godot-demo-projects/tree/master/3d/global_illumination].

Which global illumination mode should I use on meshes and lights?

Regardless of which global illumination technique you use, there is no universally "better" global illumination mode. Still, here are some recommendations for meshes:

- For static level geometry, use the **Static** global illumination mode *(default)*.
- For small dynamic geometry and players/enemies, use the **Disabled** global illumination mode. Small dynamic geometry will not be able to contribute a significant amount of indirect lighting, due to the geometry being smaller than a voxel. If you need indirect lighting for small dynamic objects, it can be simulated using an OmniLight3D or SpotLight3D node parented to the object.
- For *large* dynamic level geometry (such as a moving train), use the **Dynamic** global illumination mode. Note that this only has an effect with VoxelGI, as SDFGI and LightmapGI do not support global illumination with dynamic objects.

Here are some recommendations for light bake modes:

- For static level lighting, use the **Static** bake mode. The **Static** mode is also suitable for dynamic lights that don't change much during gameplay, such as a flickering torch.
- For short-lived dynamic effects (such as a weapon), use the **Disabled** bake mode to improve performance.
- For long-lived dynamic effects (such as a rotating alarm light), use the **Dynamic** bake mode to improve quality *(default)*. Note that this only has an effect with VoxelGI and SDFGI, as LightmapGI does not support global illumination with dynamic lights.

Alternatives to GI techniques

If none of the GI techniques mentioned above fits, it's still possible to simulate GI by placing additional lights manually. This requires more manual work, but it can offer good performance *and* good visuals if done right. This approach is still used in many modern games to this day.

When targeting low-end hardware in situations where using LightmapGI is not viable (such as procedurally generated levels), relying on environment lighting alone or a constant ambient light factor may be a necessity. This may result in flatter visuals, but adjusting the ambient light color and sky contribution still makes it possible to achieve acceptable results in most cases.

Using Voxel global illumination

VoxelGI is a form of fully real-time global illumination, intended to be used for small/medium-scale 3D scenes. VoxelGI is fairly demanding on the GPU, so it's best used when targeting dedicated graphics cards.

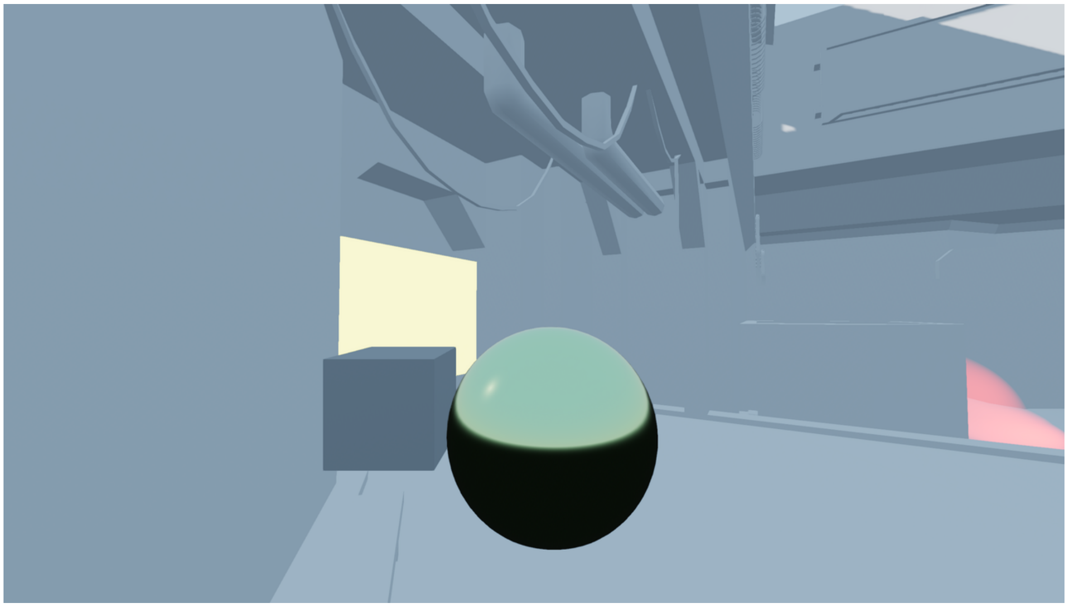
Important

VoxelGI is only supported when using the Forward Plus rendering backend, not the Forward Mobile or Compatibility backends.

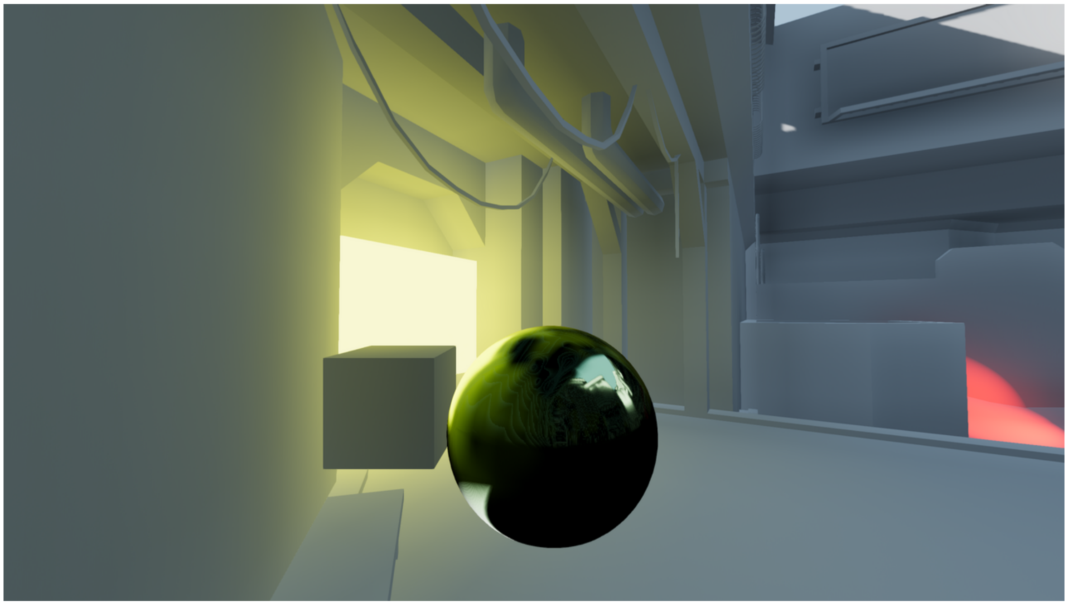
See also

Not sure if VoxelGI is suited to your needs? See Which global illumination technique should I use? for a comparison of GI techniques available in Godot 4.

Visual comparison



VoxelGI disabled.



VoxelGI enabled.

Setting up VoxelGI

1. Make sure your static level geometry is imported with the Light Baking option set to **Static** or **Static Lightmaps** in the Import

dock. For manually added MeshInstance3D nodes, make sure the **Global Illumination > Mode** property is set to **Static** in the inspector.
2. Create a VoxelGI node in the Scene tree dock.
3. Move the VoxelGI node to the center of the area you want it to cover by dragging the manipulation gizmo in the 3D viewport. Then adjust the VoxelGI's extents by dragging the red points in the 3D viewport (or enter values in the inspector). Make sure the VoxelGI's extents aren't unnecessarily large, or quality will suffer.
4. Select the VoxelGI node and click **Bake** at the top of the 3D editor viewport. This will take at least a few seconds to complete (depending on the number of VoxelGI subdivisions and scene complexity).

If at least one mesh contained within the VoxelGI's extents has its global illumination mode set to **Static**, you should see indirect lighting appear within the scene.

Note

To avoid bloating text-based scene files with large amounts of binary data, make sure the VoxelGIData resource is *always* saved to an external binary file. This file must be saved with a `.res` (binary resource) extension instead of `.tres` (text-based resource). Using an external binary resource for VoxelGIData will keep your text-based scene small while ensuring it loads and saves quickly.

VoxelGI node properties

The following properties can be adjusted in the VoxelGI node inspector before baking:

- **Subdiv:** Higher values result in more precise indirect lighting, at the cost of lower performance, longer bake times and increased storage requirements.
- **Extents:** Represents the size of the box in which indirect lighting should be baked. Extents are centered around the VoxelGI node's origin.

The following properties can be adjusted in the VoxelGIData *resource* that is contained within a VoxelGI node after it has been baked:

- **Dynamic Range:** The maximum brightness that can be represented in indirect lighting. Higher values make it possible to represent brighter indirect light, at the cost of lower precision (which can result in visible banding). If in doubt, leave this unchanged.
- **Energy:** The indirect lighting's overall energy. This also effects the energy of direct lighting emitted by meshes with emissive materials.
- **Bias:** Optional bias added to lookups into the voxel buffer at run time. This helps avoid self-occlusion artifacts.
- **Normal Bias:** Similar to **Bias**, but offsets the lookup into the voxel buffer by the surface normal. This also helps avoid self-occlusion artifacts. Higher values reduce self-reflections visible in non-rough materials, at the cost of more visible light leaking and flatter-looking indirect lighting. To prioritize hiding self-reflections over lighting quality, set **Bias** to `0.0` and **Normal Bias** to a value between `1.0` and `2.0`.
- **Propagation:** The energy factor to use for bounced indirect lighting. Higher values will result in brighter, more diffuse lighting (which may end up looking too flat). When **Use Two Bounces** is enabled, you may want to decrease **Propagation** to compensate for the overall brighter indirect lighting.
- **Use Two Bounces:** If enabled, lighting will bounce twice instead of just once. This results in more realistic-looking indirect lighting, and makes indirect lighting visible in reflections

as well. Enabling this generally has no noticeable performance cost.
- **Interior:** If enabled, environment sky lighting will not be taken into account by VoxelGI. This should be enabled in indoor scenes to avoid light leaking from the environment.

VoxelGI interaction with lights and objects

To ensure correct visuals when using VoxelGI, you must configure your meshes and lights' global illumination properties according to their *purpose* in the scene (static or dynamic).

There are 3 global illumination modes available for meshes:

- **Disabled:** The mesh won't be taken into account for VoxelGI baking. The mesh will *receive* indirect lighting from the scene, but it will not *contribute* indirect lighting to the scene.
- **Static (default):** The mesh will be taken into account for VoxelGI baking. The mesh will both receive *and* contribute indirect lighting to the scene. If the mesh is changed in any way after baking, the VoxelGI node must be baked again. Otherwise, indirect lighting will look incorrect.
- **Dynamic:** The mesh won't be taken into account for VoxelGI baking, but it will still receive *and* contribute indirect lighting to the scene in real-time. This option is much slower compared to **Static**. Only use the **Dynamic** global illumination mode on large meshes that will change significantly during gameplay.

Additionally, there are 3 bake modes available for lights (DirectionalLight3D, OmniLight3D and SpotLight3D):

- **Disabled:** The light won't be taken into account for VoxelGI baking. The light won't contribute indirect lighting to the scene.

- **Static:** The light will be taken into account for VoxelGI baking. The light will contribute indirect lighting to the scene. If the light is changed in any way after baking, the VoxelGI node must be baked again or indirect lighting will look incorrect. If in doubt, use this mode for level lighting.
- **Dynamic (default):** The light won't be taken into account for VoxelGI baking, but it will still contribute indirect lighting to the scene in real-time. This option is slower compared to **Static**. Only use the **Dynamic** global illumination mode on lights that will change significantly during gameplay.

Note

The amount of indirect energy emitted by a light depends on its color, energy *and* indirect energy properties. To make a specific light emit more or less indirect energy without affecting the amount of direct light emitted by the light, adjust the **Indirect Energy** property in the Light3D inspector.

See also

See Which global illumination mode should I use on meshes and lights? for general usage recommendations.

Adjusting VoxelGI performance and quality

Since VoxelGI is relatively demanding, it will perform best on systems with recent dedicated GPUs. On older dedicated GPUs and integrated graphics, tweaking the settings is necessary to achieve reasonable performance.

In the Project Settings' **Rendering > Global Illumination** section, VoxelGI quality can also be adjusted in two ways:

- **Voxel Gi > Quality:** If set to **Low** instead of **High**, voxel cone tracing will only use 4 taps instead of 6. This speeds up rendering at the cost of less pronounced ambient occlusion.
- **Gi > Use Half Resolution:** If enabled, both VoxelGI and SDFGI will have their GI buffer rendering at halved resolution. For instance, when rendering in 3840×2160, the GI buffer will be computed at a 1920×1080 resolution. Enabling this option saves a lot of GPU time, but it can introduce visible aliasing around thin details.

Note that the **Advanced** toggle must be enabled in the project settings dialog for the above settings to be visible.

Additionally, VoxelGI can be disabled entirely by hiding the VoxelGI node. This can be used for comparison purposes or to improve performance on low-end systems.

Reducing VoxelGI light leaks and artifacts

After baking VoxelGI, you may notice indirect light is leaking at some spots in your level geometry. This can be remedied in several ways:

- For both light leaking and artifacts, try moving or rotating the VoxelGI node then bake it again.
- To combat light leaking in general, ensure your level geometry is fully sealed. This is best done in the 3D modeling software used to design the level, but primitive MeshInstance3D nodes with their global illumination mode set to **Static** can also be used.
- To combat light leaking with thin geometry, it's recommended to make the geometry in question thicker. If this is not possible,

then add a primitive MeshInstance3D node with its global illumination mode set to **Static**. Bake VoxelGI again, then hide the primitive MeshInstance3D node (it will still be taken into account by VoxelGI). For optimal results, the MeshInstance3D should have a material whose color matches the original thin geometry.
- To combat artifacts that can appear on reflective surfaces, try increasing **Bias** and/or **Normal Bias** in the VoxelGIData resource as described above. Do not increase these values too high, or light leaking will become more pronounced.

If you notice VoxelGI nodes popping in and out of existence as the camera moves, this is most likely because the engine is rendering too many VoxelGI instances at once. Godot is limited to rendering 8 VoxelGI nodes at once, which means up to 8 instances can be in the camera view before some of them will start flickering.

Additionally, for performance reasons, Godot can only blend between 2 VoxelGI nodes at a given pixel on the screen. If you have more than 2 VoxelGI nodes overlapping, global illumination may appear to flicker as the camera moves or rotates.

Signed distance field global illumination (SDFGI)

Signed distance field global illumination (SDFGI) is a novel technique available in Godot 4.0. It provides semi-real-time global illumination that scales to any world size and works with procedurally generated levels.

SDFGI supports dynamic lights, but *not* dynamic occluders or dynamic emissive surfaces. Therefore, SDFGI provides better real-time ability than baked lightmaps, but worse real-time ability than VoxelGI.

From a performance standpoint, SDFGI is one of the most demanding global illumination techniques in Godot. Like with VoxelGI, there are still many settings available to tweak its performance requirements at the cost of quality.

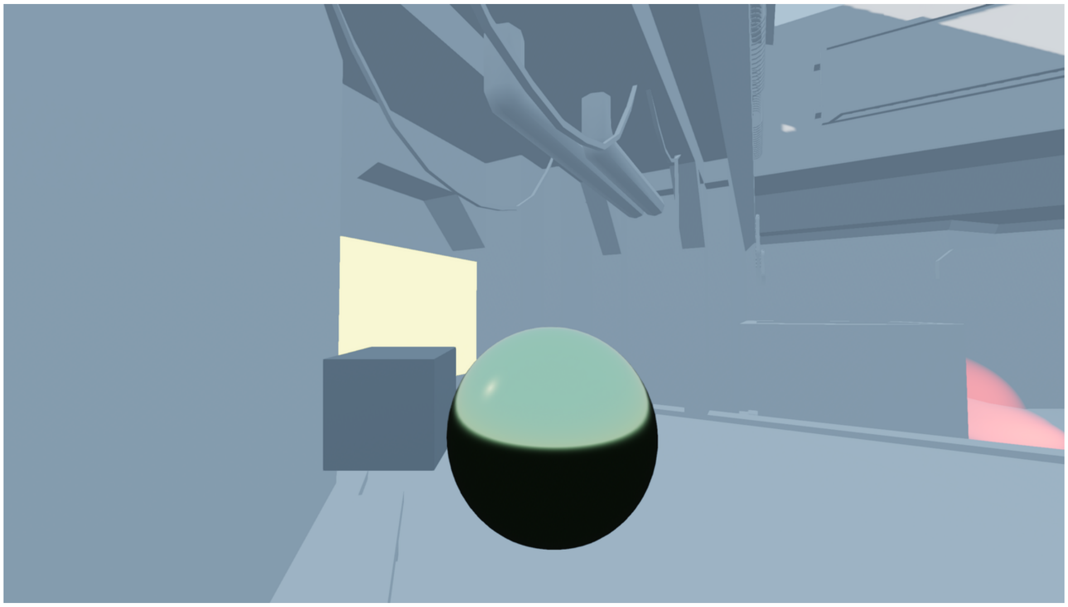
Important

SDFGI is only supported when using the Forward Plus rendering backend, not the Forward Mobile or Compatibility backends.

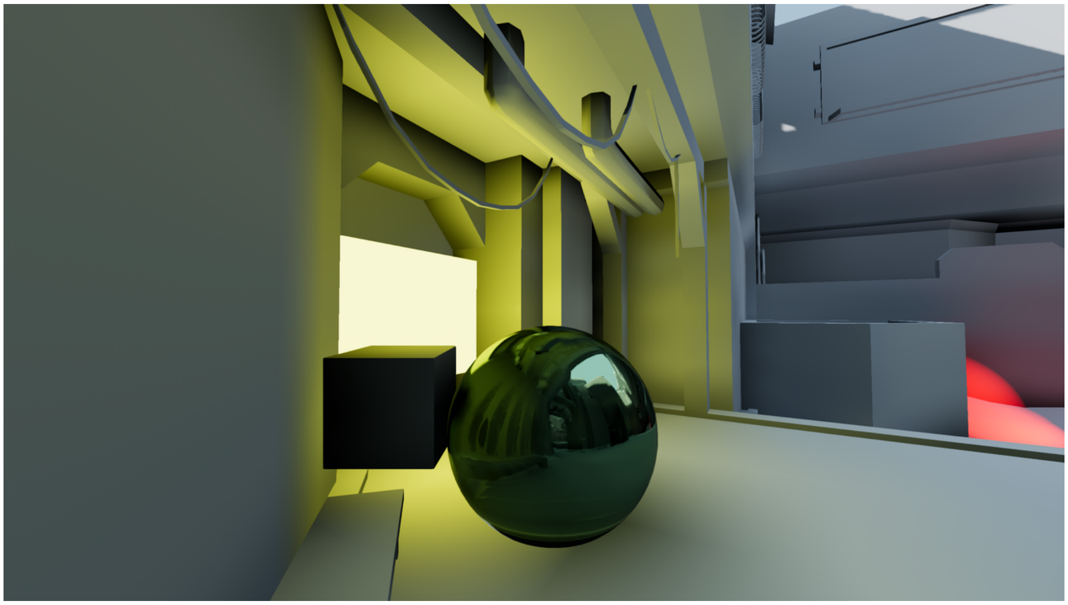
See also

Not sure if SDFGI is suited to your needs? See Which global illumination technique should I use? for a comparison of GI techniques available in Godot 4.

Visual comparison



SDFGI disabled.



SDFGI enabled.

Setting up SDFGI

In Godot, SDFGI is the global illumination technique with the fewest required steps to enable:

1. Make sure your MeshInstance nodes have their **Global Illumination > Mode** property set to **Static** in the inspector.

    - For imported 3D scenes, the bake mode can be configured in the Import dock after selecting the 3D scene file in the FileSystem dock.

2. Add a WorldEnvironment node and create an Environment resource for it.
3. Edit the Environment resource, scroll down to the **SDFGI** section and unfold it.
4. Enable **SDFGI > Enabled**. SDFGI will automatically follow the camera when it moves, so you do not need to configure extents (unlike VoxelGI).

Environment SDFGI properties

In the Environment resource, there are several properties available to adjust SDFGI appearance and quality:

- **Use Occlusion:** If enabled, SDFGI will throw additional rays to find and reduce light leaks. This has a performance cost, so only enable this property if you actually need it.
- **Read Sky Light:** If enabled, the environment lighting is represented in the global illumination. This should be enabled in outdoor scenes and disabled in fully indoor scenes.
- **Bounce Feedback:** By default, indirect lighting only bounces once when using SDFGI. Setting this value above `0.0` will cause SDFGI to bounce more than once, which provides more realistic indirect lighting at a small performance cost. Sensible values are usually between `0.3` and `1.0` depending on the scene. Note that in some scenes, values above `0.5` can cause infinite feedback loops to happen, causing the scene to become extremely bright in a few seconds' time. If your indirect lighting looks "splotchy", consider increasing this value above `0.0` to get more uniform-

looking lighting. If your lighting ends up looking too bright as a result, decrease **Energy** to compensate.

- **Cascades:** Higher values result in more detailed GI information (and/or greater maximum distance), but are significantly more expensive on the CPU and GPU. The performance cost of having more cascades especially increases when the camera moves fast, so consider decreasing this to `4` or lower if your camera moves fast.
- **Min Cell Size:** The minimum SDFGI cell size to use for the nearest, most detailed cascade. Lower values result in more accurate indirect lighting and reflection at the cost of lower performance. Adjusting this setting also affects **Cascade 0 Distance** and **Max Distance** automatically.
- **Cascade 0 Distance:** The distance at which the nearest, most detailed cascade ends. Greater values make the nearest cascade transition less noticeable, at the cost of reducing the level of detail in the nearest cascade. Adjusting this setting also affects **Min Cell Size** and **Max Distance** automatically.
- **Max Distance:** Controls how far away the signed distance field will be computed (for the least detailed cascade). SDFGI will not have any effect past this distance. This value should always be set below the Camera's Far value, as there is no benefit in computing SDFGI past the viewing distance. Adjusting this setting also affects **Min Cell Size** and **Cascade 0 Distance** automatically.
- **Y Scale:** Controls how far apart SDFGI probes are spread *vertically*. By default, vertical spread is the same as horizontal. However, since most game scenes aren't highly vertical, setting the Y Scale to `75%` or even `50%` can provide better quality and reduce light leaks without impacting performance.
- **Energy:** The brightness multiplier for SDFGI's indirect lighting.
- **Normal Bias:** The normal bias to use for SDFGI's probe ray bounces. Unlike **Probe Bias**, this only increases the value in relation to the mesh's normals. This makes the bias adjustment more nuanced and avoids increasing the bias too much for no

reason. Increase this value if you notice striping artifacts in indirect lighting or reflections.
- **Probe Bias:** The bias to use for SDFGI's probe ray bounces. Increase this value if you notice striping artifacts in indirect lighting or reflections.

SDFGI interaction with lights and objects

The amount of indirect energy emitted by a light is governed by its color, energy *and* indirect energy properties. To make a specific light emit more or less indirect energy without affecting the amount of direct light emitted by the light, adjust the **Indirect Energy** property in the Light3D inspector.

To ensure correct visuals when using SDFGI, you must configure your meshes and lights' global illumination properties according to their *purpose* in the scene (static or dynamic).

There are 3 global illumination modes available for meshes:

- **Disabled:** The mesh won't be taken into account in SDFGI generation. The mesh will receive indirect lighting from the scene, but it will not contribute indirect lighting to the scene.
- **Static (default):** The mesh will be taken into account in SDFGI generation. The mesh will both receive *and* contribute indirect lighting to the scene. If the mesh is changed in any way after SDFGI is generated, the camera must move away from the object then move back close to it for SDFGI to regenerate. Alternatively, SDFGI can be toggled off and back on. If neither is done, indirect lighting will look incorrect.
- **Dynamic (not supported with SDFGI):** The mesh won't be taken into account in SDFGI generation. The mesh will receive indirect lighting from the scene, but it will not contribute indirect

lighting to the scene. _This acts identical to the **Disabled*_ bake mode when using SDFGI.*

Additionally, there are 3 bake modes available for lights (DirectionalLight3D, OmniLight3D and SpotLight3D):

- **Disabled:** The light won't be taken into account for SDFGI baking. The light won't contribute indirect lighting to the scene.
- **Static:** The light will be taken into account for SDFGI baking. The light will contribute indirect lighting to the scene. If the light is changed in any way after baking, indirect lighting will look incorrect until the camera moves away from the light and back (which causes SDFGI to be baked again). will look incorrect. If in doubt, use this mode for level lighting.
- **Dynamic (default):** The light won't be taken into account for SDFGI baking, but it will still contribute indirect lighting to the scene in real-time. This option is slower compared to **Static**. Only use the **Dynamic** global illumination mode on lights that will change significantly during gameplay.

Note

The amount of indirect energy emitted by a light depends on its color, energy *and* indirect energy properties. To make a specific light emit more or less indirect energy without affecting the amount of direct light emitted by the light, adjust the **Indirect Energy** property in the Light3D inspector.

See also

See Which global illumination mode should I use on meshes and lights? for general usage recommendations.

Adjusting SDFGI performance and quality

Since SDFGI is relatively demanding, it will perform best on systems with recent dedicated GPUs. On older dedicated GPUs and integrated graphics, tweaking the settings is necessary to achieve reasonable performance.

In the Project Settings' **Rendering > Global Illumination** section, SDFGI quality can also be adjusted in several ways:

- **Sdfgi > Probe Ray Count:** Higher values result in better quality, at the cost of higher GPU usage. If this value is set too low, this can cause surfaces to have visible "splotches" of indirect lighting on them due to the number of rays thrown being very low.
- **Sdfgi > Frames To Converge:** Higher values result in better quality, but GI will take more time to fully converge. The effect of this setting is especially noticeable when first loading a scene, or when lights with a bake mode other than **Disabled** are moving fast. If this value is set too low, this can cause surfaces to have visible "splotches" of indirect lighting on them due to the number of rays thrown being very low. If your scene's lighting doesn't have fast-moving lights that contribute to GI, consider setting this to `30` to improve quality without impacting performance.
- **Sdfgi > Frames To Update Light:** Lower values result in moving lights being reflected faster, at the cost of higher GPU usage. If your scene's lighting doesn't have fast-moving lights that contribute to GI, consider setting this to `16` to improve performance.
- **Gi > Use Half Resolution:** If enabled, both SDFGI and VoxelGI will have their GI buffer rendering at halved resolution. For instance, when rendering in 3840×2160, the GI buffer will

be computed at a 1920×1080 resolution. Enabling this option saves a lot of GPU time, but it can introduce visible aliasing around thin details.

SDFGI rendering performance also depends on the number of cascades and the cell size chosen in the Environment resource (see above).

SDFGI caveats

SDFGI has some downsides due to its cascaded nature. When the camera moves, cascade shifts may be visible in indirect lighting. This can be alleviated by adjusting the cascade size, but also by adding fog (which will make distant cascade shifts less noticeable).

Additionally, performance will suffer if the camera moves too fast. This can be fixed in two ways:

- Ensuring the camera doesn't move too fast in any given situation.
- Temporarily disabling SDFGI in the Environment resource if the camera needs to be moved at a high speed, then enabling SDFGI once the camera speed slows down.

When SDFGI is enabled, it will also take some time for global illumination to be fully converged (25 frames by default). This can create a noticeable transition effect while GI is still converging. To hide this, you can use a ColorRect node that spans the whole viewport and fade it out when switching scenes using an AnimationPlayer node.

The signed distance field is only updated when the camera moves in and out of a cascade. This means that if geometry is modified in the distance, the global illumination appearance will be correct once the camera gets closer. However, if a nearby object with a bake mode set to **Static** or **Dynamic** is moved (such as a door), the global illumination will appear incorrect until the camera moves away from the object.

SDFGI's sharp reflections are only visible on opaque materials. Transparent materials will only use rough reflections, even if the material's roughness is lower than 0.2.

Using Lightmap global illumination

Baked lightmaps are a workflow for adding indirect (or fully baked) lighting to a scene. Unlike the VoxelGI and SDFGI approaches, baked lightmaps work fine on low-end PCs and mobile devices, as they consume almost no resources at run-time. Also unlike VoxelGI and SDFGI, baked lightmaps can optionally be used to store direct lighting, which provides even further performance gains.

Unlike VoxelGI and SDFGI, baked lightmaps are completely static. Once baked, they can't be modified at all. They also don't provide the scene with reflections, so using Reflection probes together with it on interiors (or using a Sky on exteriors) is a requirement to get good quality.

As they are baked, they have fewer problems than VoxelGI and SDFGI regarding light bleeding, and indirect light will often look better. The downside is that baking lightmaps takes longer compared to baking VoxelGI. While baking VoxelGI can be done in a matter of seconds, baking lightmaps can take several minutes if not more. This can slow down iteration speed significantly, so it is recommended to bake lightmaps only when you actually need to see changes in lighting. Since Godot 4.0, lightmaps are baked on the GPU, making light baking faster if you have a mid-range or high-end dedicated GPU.

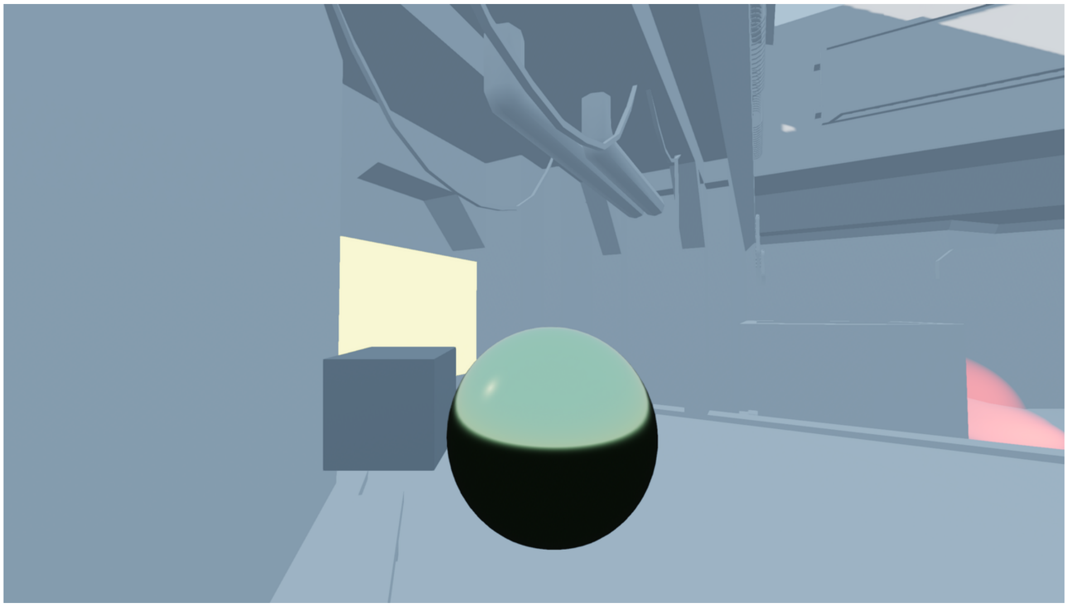
Baking lightmaps will also reserve baked materials' UV2 slot, which means you can no longer use it for other purposes in materials (either in the built-in Standard Material 3D and ORM Material 3D or in custom shaders).

Despite their lack of flexibility, baked lightmaps typically offer both the best quality *and* performance at the same time in (mostly) static scenes. This makes lightmaps still popular in game development, despite lightmaps being the oldest technique for global illumination in video games.

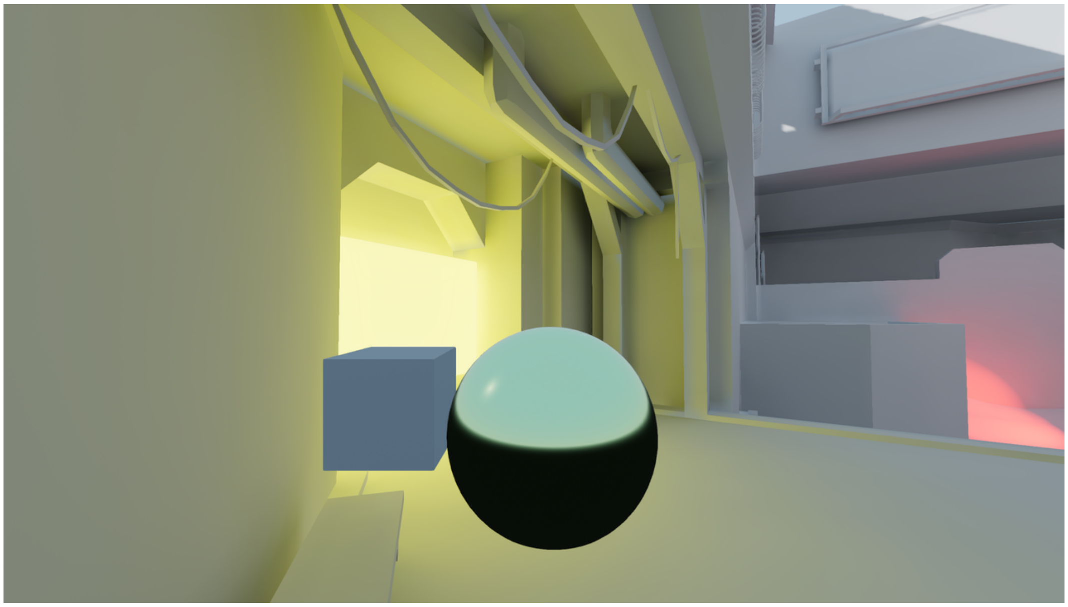
See also

Not sure if LightmapGI is suited to your needs? See Which global illumination technique should I use? for a comparison of GI techniques available in Godot 4.

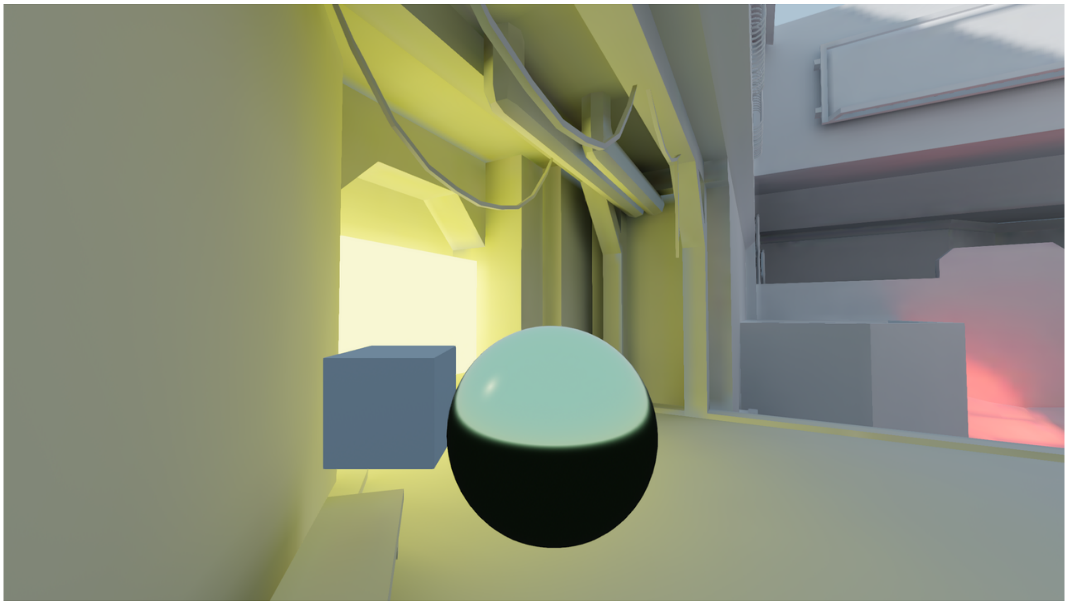
Visual comparison



LightmapGI disabled.

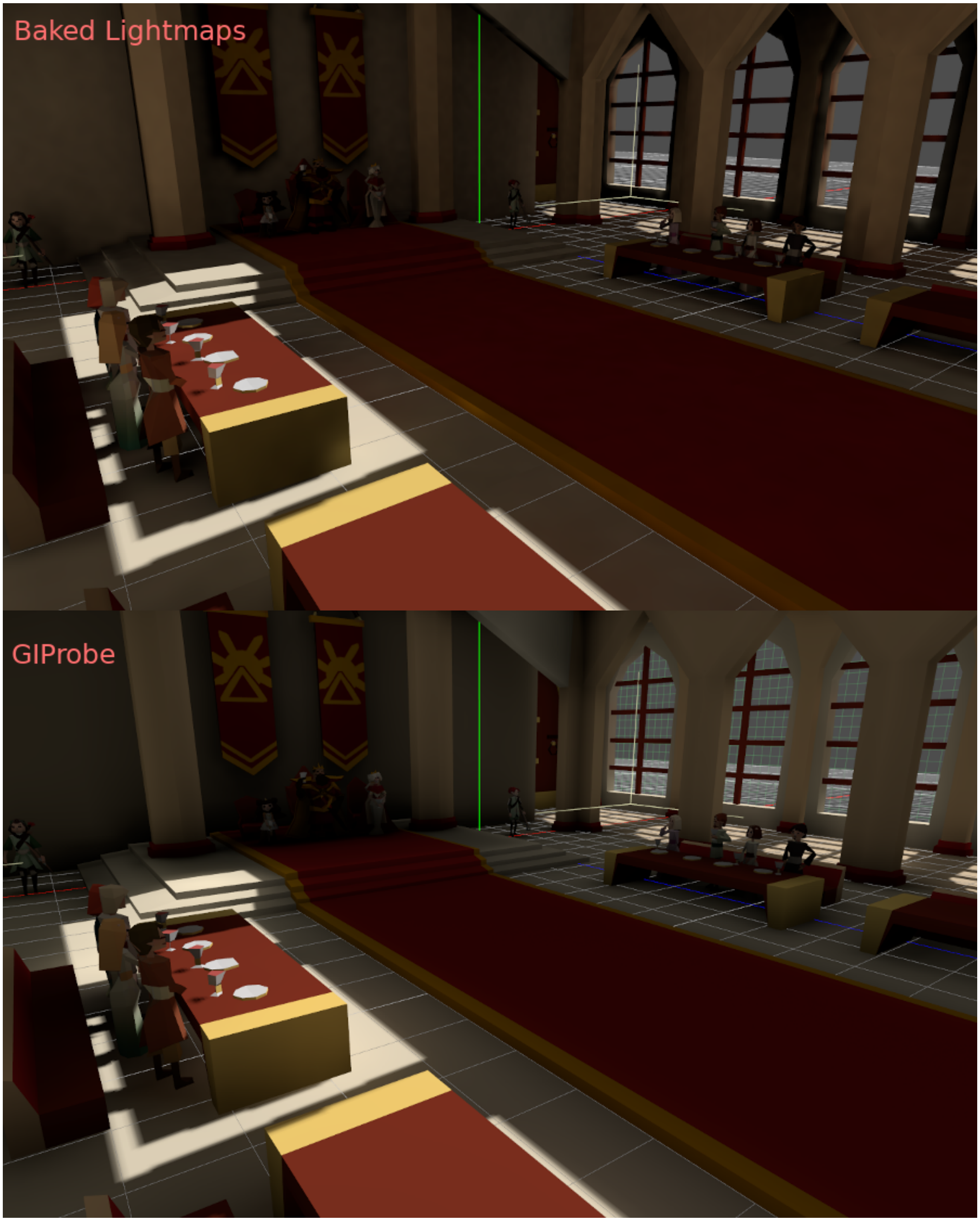


LightmapGI enabled (with indirect light baked only). Direct light is still real-time, allowing for subtle changes during gameplay.



LightmapGI enabled (with direct and indirect light baked). Best performance, but lower quality visuals. Notice the blurrier sun shadow in the top-right corner.

Here are some comparisons of how LightmapGI vs. VoxelGI look. Notice that lightmaps are more accurate, but also suffer from the fact that lighting is on an unwrapped texture, so transitions and resolution may not be that good. VoxelGI looks less accurate (as it's an approximation), but smoother overall.


Baked Lightmaps
GIProbe

SDFGI is also less accurate compared to LightmapGI. However, SDFGI can support large open worlds without any need for baking.

Setting up

Warning

Baking lightmaps in the Android and web editors is not supported due to graphics API limitations on those devices. On Android and web platforms, only *rendering* lightmaps that were baked on a desktop PC is supported.

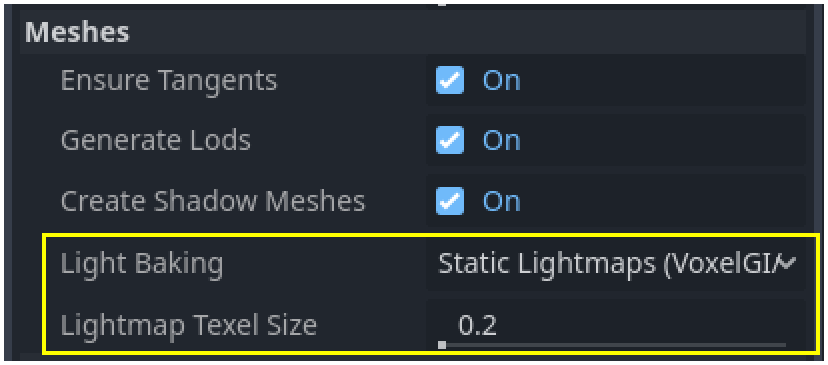
First of all, before the lightmapper can do anything, the objects to be baked need a UV2 layer and a texture size. A UV2 layer is a set of secondary texture coordinates that ensures any face in the object has its own place in the UV map. Faces must not share pixels in the texture.

There are a few ways to ensure your object has a unique UV2 layer and texture size:

Unwrap on scene import (recommended)

In most scenarios, this is the best approach to use. The only downside is that, on large models, unwrapping can take a while on import. Nonetheless, Godot will cache the UV2 across reimports, so it will only be regenerated when needed.

Select the imported scene in the filesystem dock, then go to the **Import** dock. There, the following option can be modified:



Meshes

Ensure Tangents ☑ On

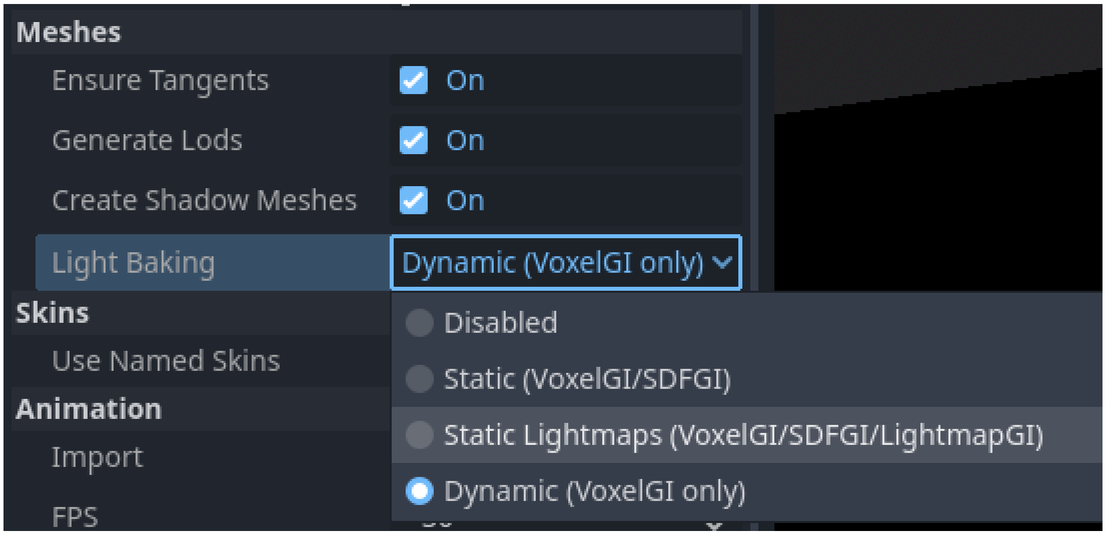
Generate Lods ☑ On

Create Shadow Meshes ☑ On

Light Baking: Static Lightmaps (VoxelGI/

Lightmap Texel Size: 0.2

The **Meshes > Light Baking** option must be set to **Static Lightmaps (VoxelGI/SDFGI/LightmapGI)**:



Meshes

Ensure Tangents ☑ On

Generate Lods ☑ On

Create Shadow Meshes ☑ On

Light Baking: Dynamic (VoxelGI only)

Skins
Use Named Skins
Animation
Import
FPS
○ Disabled
○ Static (VoxelGI/SDFGI)
○ Static Lightmaps (VoxelGI/SDFGI/LightmapGI)
● Dynamic (VoxelGI only)

When unwrapping on import, you can adjust the texture size using the **Meshes > Lightmap Texel Size** option. *Lower* values will result in more detailed lightmaps, possibly resulting in higher visual quality at the cost of longer bake times and larger lightmap file sizes. The default value of `0.2` is suited for small/medium-sized scenes, but you may want to increase it to `0.5` or even more for larger scenes. This is especially the case if you're baking indirect lighting only, as indirect light is low-frequency data (which means it doesn't need high-resolution textures to be accurately represented).

The effect of setting this option is that all meshes within the scene will have their UV2 maps properly generated.

Warning

When reusing a mesh within a scene, keep in mind that UVs will be generated for the first instance found. If the mesh is re-used with different scales (and the scales are wildly different, more than half or twice), this will result in inefficient lightmaps. To avoid this, adjust the **Lightmap Scale** property in the GeometryInstance3D section of a MeshInstance3D node. This lets you *increase* the level of lightmap detail for specific MeshInstance3D nodes (but not decrease it).

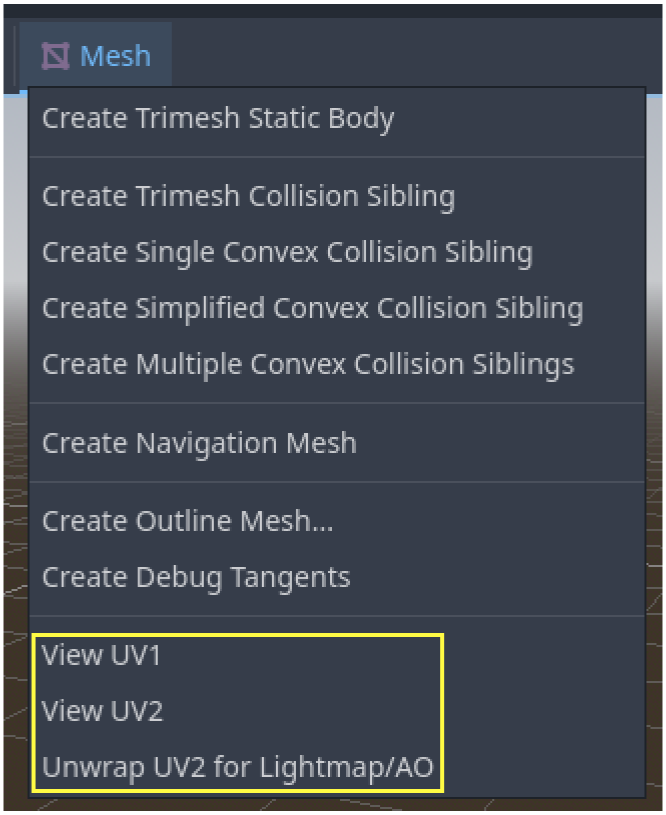
Also, the `*.unwrap_cache` files should *not* be ignored in version control as these files guarantee that UV2 reimports are consistent across platforms and engine versions.

Unwrap from within Godot

Warning

If this Mesh menu operation is used on an imported 3D scene, the generated UV2 will be lost when the scene is reloaded.

Godot has an option to unwrap meshes and visualize the UV channels. After selecting a MeshInstance3D node, it can be found in the **Mesh** menu at the top of the 3D editor viewport:

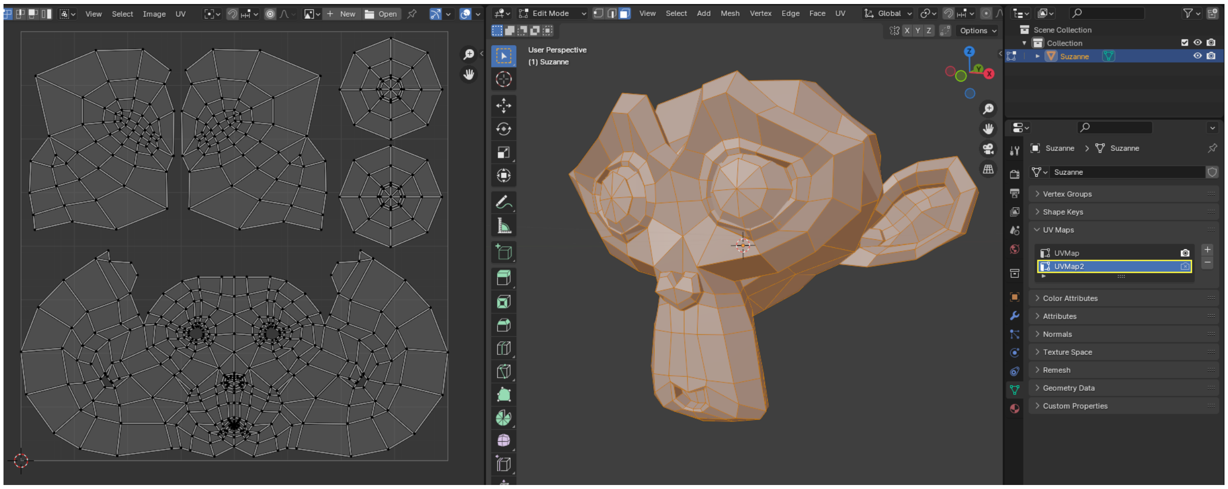

Mesh
Create Trimesh Static Body
Create Trimesh Collision Sibling
Create Single Convex Collision Sibling
Create Simplified Convex Collision Sibling
Create Multiple Convex Collision Siblings
Create Navigation Mesh
Create Outline Mesh...
Create Debug Tangents
View UV1
View UV2
Unwrap UV2 for Lightmap/AO

This will generate a second set of UV2 coordinates which can be used for baking. It will also set the texture size automatically.

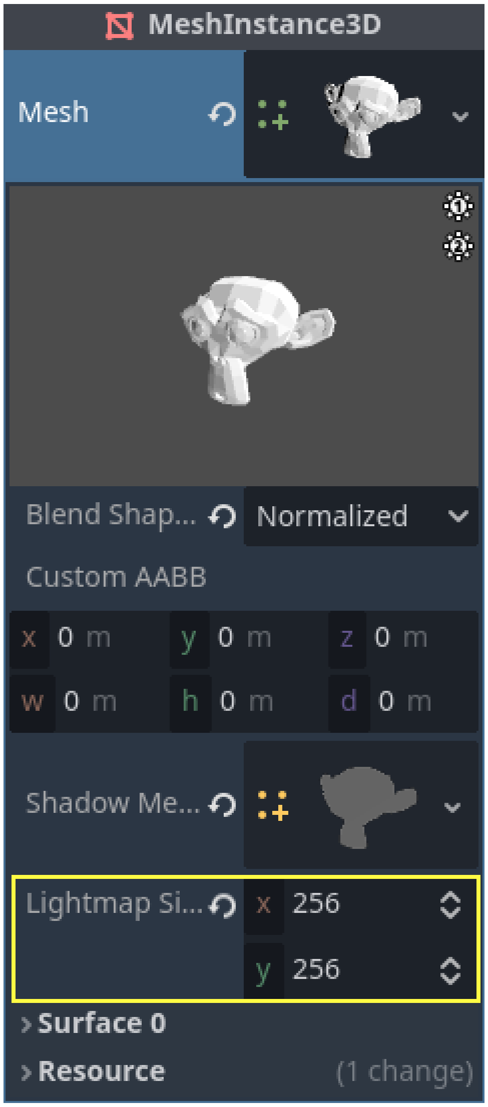
Unwrap from your 3D modeling software

The last option is to do it from your favorite 3D app. This approach is generally **not recommended**, but it's explained so that you know it exists. The main advantage is that, on complex objects that you may want to re-import a lot, the texture generation process can be quite costly within Godot, so having it unwrapped before import can be faster.

Simply do an unwrap on the second UV2 layer.


View
Select
Image
UV
New
Open
Edit Mode
View
Select
Add
Mesh
Vertex
Edge
Face
UV
Global
Options
User Perspective
(1) Suzanne
Scene Collection
Collection
Suzanne
Suzanne
Suzanne
Suzanne
Vertex Groups
Shape Keys
UV Maps
UVMap
UVMap2
Color Attributes
Attributes
Normals
Texture Space
Remesh
Geometry Data
Custom Properties

Then import the 3D scene normally. Remember you will need to set the texture size on the mesh after import.


MeshInstance3D
Mesh
Blend Shap...
Normalized
Custom AABB
x 0 m
y 0 m
z 0 m
w 0 m
h 0 m
d 0 m
Shadow Me...
Lightmap Si...
x 256
y 256
Surface 0
Resource
(1 change)

If you use external meshes on import, the size will be kept. Be wary that most unwrappers in 3D modeling software are not quality-oriented, as they are meant to work quickly. You will mostly need to use seams or other techniques to create better unwrapping.

Generating UV2 for primitive meshes

Note

This option is only available for primitive meshes such as BoxMesh, CylinderMesh, PlaneMesh, etc.

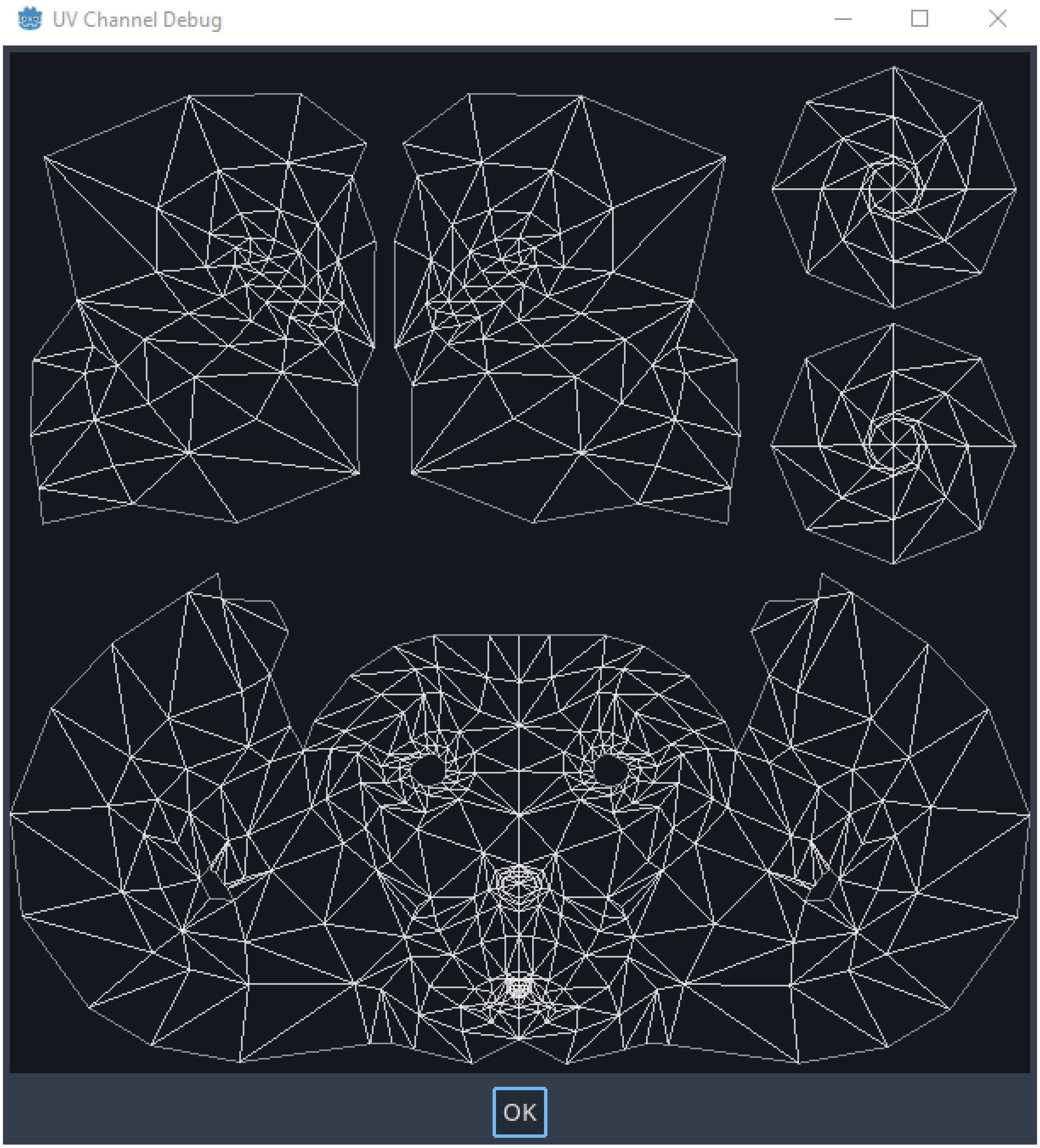
Enabling UV2 on primitive meshes allows you to make them receive and contribute to baked lighting. This can be used in certain lighting setups. For instance, you could hide a torus that has an emissive material after baking lightmaps to create an area light that follows the shape of a torus.

By default, primitive meshes do not have UV2 generated to save resources (as these meshes may be created during gameplay). You can edit a primitive mesh in the inspector and enable **Add UV2** to make the engine procedurally generate UV2 for a primitive mesh. The default **UV2 Padding** value is tuned to avoid most lightmap bleeding, without wasting too much space on the edges. If you notice lightmap bleeding on a specific primitive mesh only, you may have to increase **UV2 Padding**.

Lightmap Size Hint represents the size taken by a single mesh on the lightmap texture, which varies depending on the mesh's size properties and the **UV2 Padding** value. **Lightmap Size Hint** should not be manually changed, as any modifications will be lost when the scene is reloaded.

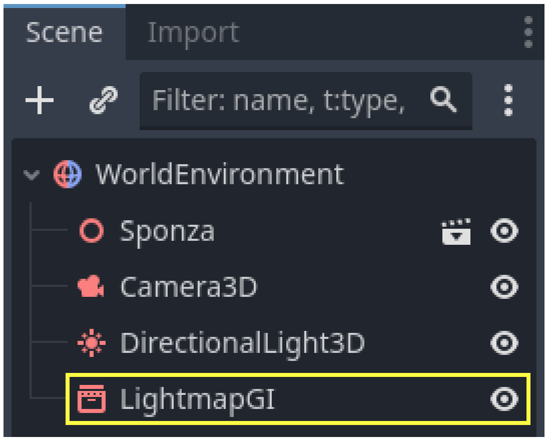
Checking UV2

In the **Mesh** menu mentioned before, the UV2 texture coordinates can be visualized. If something is failing, double-check that the meshes have these UV2 coordinates:



Setting up the scene

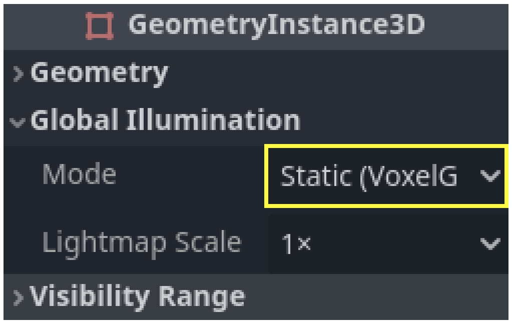
Before anything is done, a **LightmapGI** node needs to be added to a scene. This will enable light baking on all nodes (and sub-nodes) in that scene, even on instanced scenes.


Scene
Import
Filter: name, t:type,
WorldEnvironment
Sponza
Camera3D
DirectionalLight3D
LightmapGI

A sub-scene can be instanced several times, as this is supported by the baker. Each instance will be assigned a lightmap of its own. To avoid issues with inconsistent lightmap texel scaling, make sure to respect the rule about mesh scaling mentioned before.

Setting up meshes

For a **MeshInstance3D** node to take part in the baking process, it needs to have its bake mode set to **Static**. Meshes that have their bake mode set to **Disabled** or **Dynamic** will be ignored by the lightmapper.


GeometryInstance3D
Geometry
Global Illumination
Visibility Range

Mode Static (VoxelG

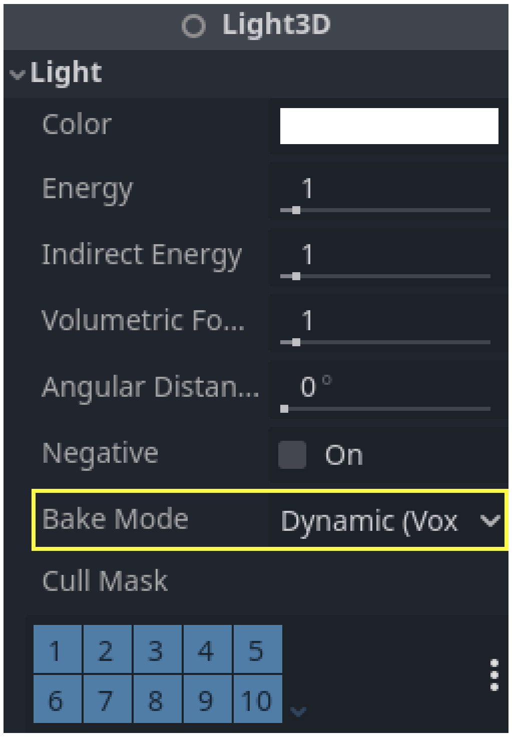
Lightmap Scale 1×

When auto-generating lightmaps on scene import, this is enabled automatically.

Setting up lights

Lights are baked with indirect light only by default. This means that shadowmapping and lighting are still dynamic and affect moving objects, but light bounces from that light will be baked.

Lights can be disabled (no bake) or be fully baked (direct and indirect). This can be controlled from the **Bake Mode** menu in lights:



Light3D

Light

Color

Energy 1

Indirect Energy 1

Volumetric Fo... 1

Angular Distan... 0°

Negative ☐ On

Bake Mode Dynamic (Vox

Cull Mask

The modes are:

Disabled

The light is ignored when baking lightmaps. This is the mode to use for dynamic lighting effects such as explosions and weapon effects.

Warning

Hiding a light has no effect on the resulting lightmap bake. This means you must use the Disabled bake mode instead of hiding the

Light node by disabling its **Visible** property.

Dynamic

This is the default mode, and is a compromise between performance and real-time friendliness. Only indirect lighting will be baked. Direct light and shadows are still real-time, as they would be without LightmapGI.

This mode allows performing *subtle* changes to a light's color, energy and position while still looking fairly correct. For example, you can use this to create flickering static torches that have their indirect light baked.

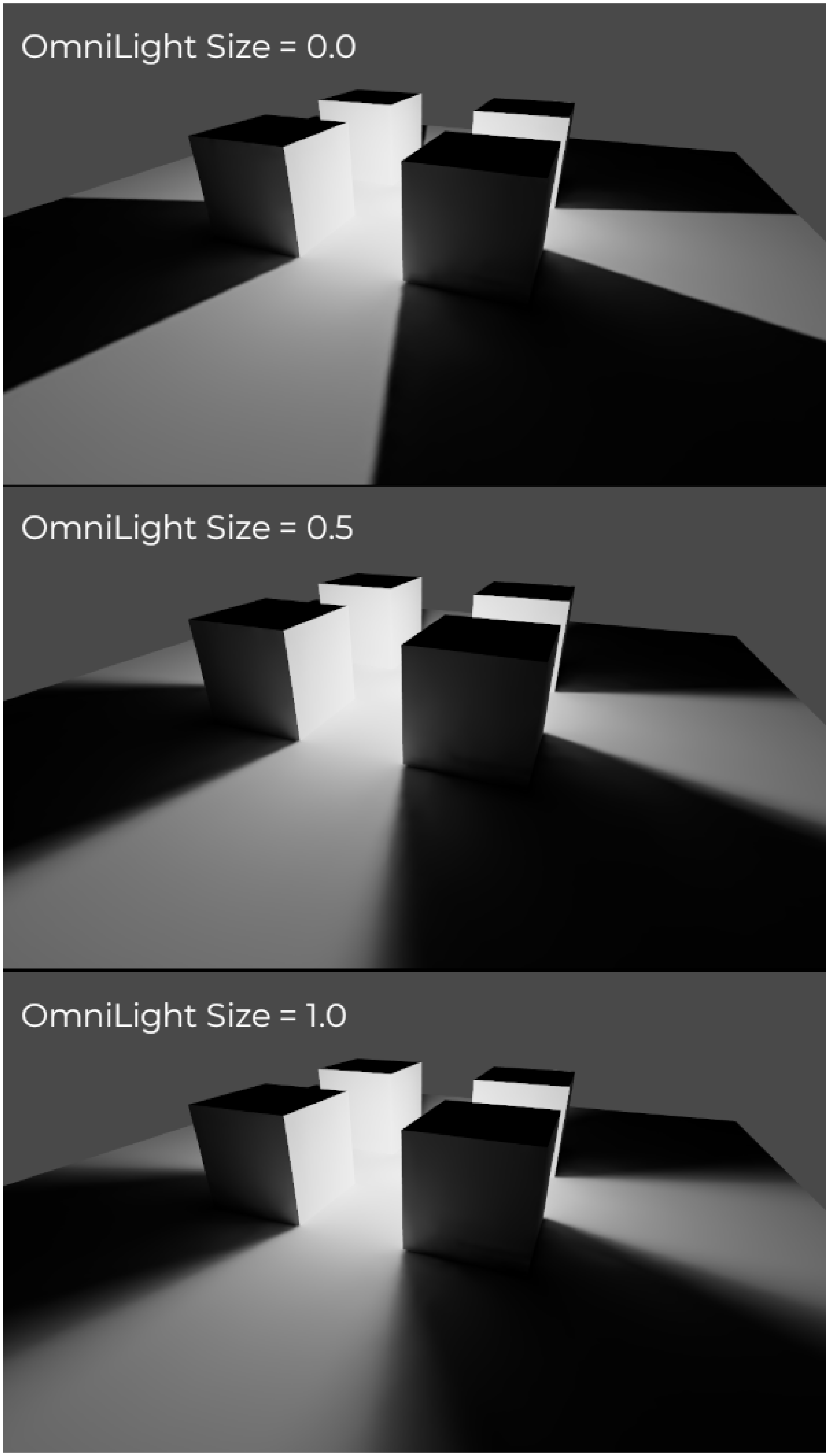
Static

Both indirect and direct lighting will be baked. Since static surfaces can skip lighting and shadow computations entirely, this mode provides the best performance along with smooth shadows that never fade based on distance. The real-time light will not affect baked surfaces anymore, but it will still affect dynamic objects. When using the **All** bake mode on a light, dynamic objects will not cast real-time shadows onto baked surfaces, so you need to use a different approach such as blob shadows instead. Blob shadows can be implemented with a Decal node.

The light will not be adjustable at all during gameplay. Moving the light or changing its color (or energy) will not have any effect on static surfaces.

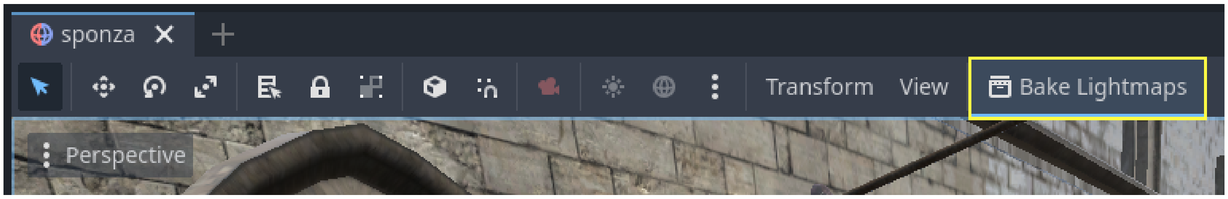
Since bake modes can be adjusted on a per-light basis, it is possible to create hybrid baked light setups. One popular option is to use a real-time DirectionalLight with its bake mode set to **Dynamic**, and use the **Static** bake mode for OmniLights and SpotLights. This provides good performance while still allowing dynamic objects to cast real-time shadows in outdoor areas.

Fully baked lights can also make use of light nodes' **Size** (omni/spot) or **Angular Distance** (directional) properties. This allows for shadows with realistic penumbra that increases in size as the distance between the caster and the shadow increases. This also has a lower performance cost compared to real-time PCSS shadows, as only dynamic objects have real-time shadows rendered on them.


OmniLight Size = 0.0
OmniLight Size = 0.5
OmniLight Size = 1.0

Baking

To begin the bake process, click the **Bake Lightmaps** button at the top of the 3D editor viewport when selecting the LightmapGI node:



This can take from seconds to minutes (or hours) depending on scene size, bake method and quality selected.

Warning

Baking lightmaps is a process that can require a lot of video memory, especially if the resulting texture is large. Due to internal limitations, the engine may also crash if the generated texture size is too large (even on systems with a lot of video memory).

To avoid crashes, make sure the lightmap texel size in the Import dock is set to a high enough value.

Tweaks

- **Quality:** Four bake quality modes are provided: Low, Medium, High, and Ultra. Higher quality takes more time, but result in a better-looking lightmap with less noise. The difference is especially noticeable with emissive materials or areas that get little to no direct lighting. Each bake quality mode can be further adjusted in the Project Settings.
- **Bounces:** The number of bounces to use for indirect lighting. The default value (`3`) is a good compromise between bake times

and quality. Higher values will make light bounce around more times before it stops, which makes indirect lighting look smoother (but also possibly brighter depending on materials and geometry).

- **Bounce Indirect Energy:** The global multiplier to use when baking lights' indirect energy. This multiplies each light's own **Indirect Energy** value. Values different from `1.0` are not physically accurate, but can be used for artistic effect.
- **Directional:** If enabled, stores directional information for lightmaps. This improves normal mapped materials' appearance for baked surfaces, especially with fully baked lights (since they also have direct light baked). The downside is that directional lightmaps are slightly more expensive to render. They also require more time to bake and result in larger file sizes.
- **Interior:** If enabled, environment lighting will not be sourced. Use this for purely indoor scenes to avoid light leaks.
- **Use Texture for Bounces:** If enabled, a texture with the lighting information will be generated to speed up the generation of indirect lighting at the cost of some accuracy. The geometry might exhibit extra light leak artifacts when using low resolution lightmaps or UVs that stretch the lightmap significantly across surfaces. Leave this enabled if unsure.
- **Use Denoiser:** If enabled, uses a denoising algorithm to make the lightmap significantly less noisy. This increases bake times and can occasionally introduce artifacts, but the result is often worth it. See Denoising for more information.
- **Denoiser Strength:** The strength of denoising step applied to the generated lightmaps. Higher values are more effective at removing noise, but can reduce shadow detail for static shadows. Only effective if denoising is enabled and the denoising method is JNLM (OIDN does not have a denoiser strength setting).
- **Bias:** The offset value to use for shadows in 3D units. You generally don't need to change this value, except if you run into issues with light bleeding or dark spots in your lightmap after

baking. This setting does not affect real-time shadows casted on baked surfaces (for lights with **Dynamic** bake mode).
- **Max Texture Size:** The maximum texture size for the generated texture atlas. Higher values will result in fewer slices being generated, but may not work on all hardware as a result of hardware limitations on texture sizes. Leave this at its default value of `16384` if unsure.
- **Environment > Mode:** Controls how environment lighting is sourced when baking lightmaps. The default value of **Scene** is suited for levels with visible exterior parts. For purely indoor scenes, set this to **Disabled** to avoid light leaks and speed up baking. This can also be set to **Custom Sky** or **Custom Color** to use environment lighting that differs from the actual scene's environment sky.
- **Gen Probes > Subdiv:** See Dynamic objects.
- **Data > Light Data:** See Lightmap data.

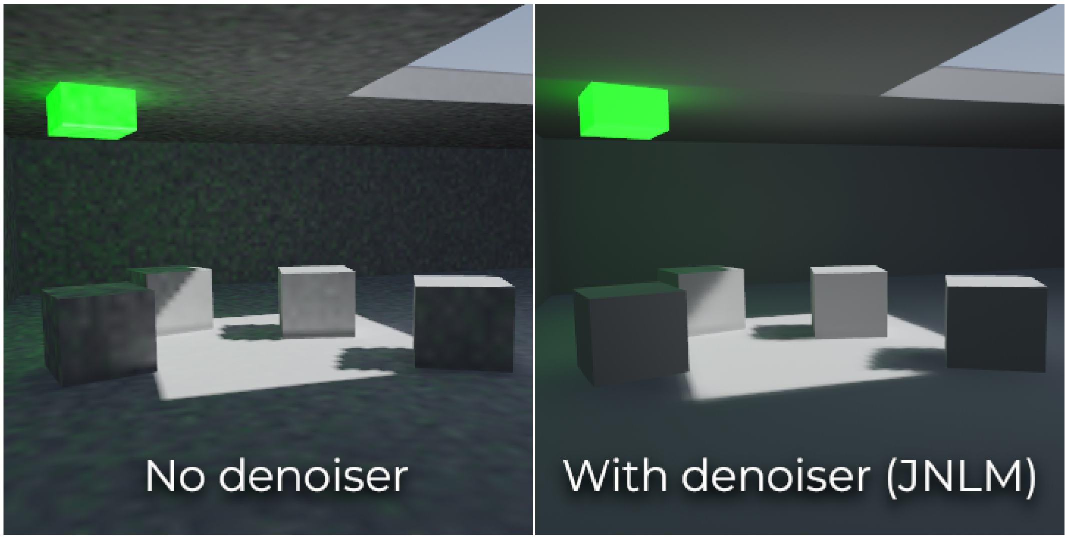
Balancing bake times with quality

Since high-quality bakes can take very long (up to dozens of minutes for large complex scenes), it is recommended to use lower quality settings at first. Then, once you are confident with your scene's lighting setup, raise the quality settings and perform a "final" bake before exporting your project.

Reducing the lightmap resolution by increasing **Lightmap Texel Size** on the imported 3D scenes will also speed up baking significantly. However, this will require you to reimport all lightmapped 3D scenes before you can bake lightmaps again.

Denoising

Since baking lightmaps relies on raytracing, there will always be visible noise in the "raw" baked lightmap. Noise is especially visible in areas that are difficult to reach by bounced light, such as indoor areas with small openings where the sunlight can enter. Noise can be reduced by increasing bake quality, but doing so will increase bake times significantly.



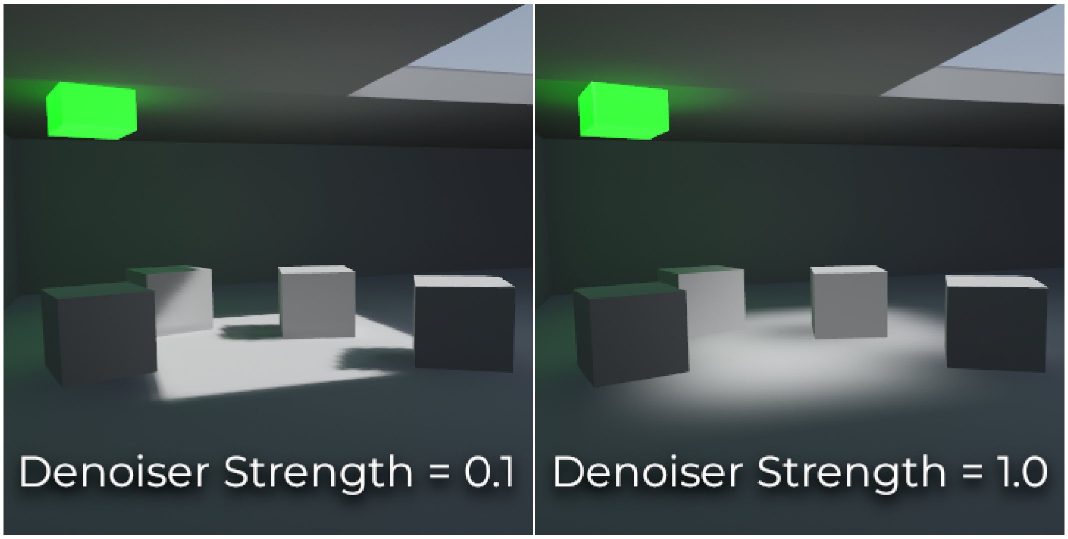
Comparison between denoising disabled and enabled (with the default JNLM denoiser).

To combat noise without increasing bake times too much, a denoiser can be used. A denoiser is an algorithm that runs on the final baked lightmap, detects patterns of noise and softens them while attempting to best preserve detail. Godot offers two denoising algorithms:

JNLM (Non-Local Means with Joint Filtering)

JNLM is the default denoising method and is included in Godot. It uses a simple but efficient denoising algorithm known as *non-local means*. JNLM runs on the GPU using a compute shader, and is compatible with any GPU that can run Godot 4's Vulkan-based rendering methods. No additional setup is required.

JNLM's denoising can be adjusted using the **Denoiser Strength** property that is visible when **Use Denoiser** enabled. Higher values can be more effective at removing noise, at the cost of suppressing shadow detail for static shadows.



Comparison between JNLM denoiser strength values.
Higher values can reduce detail.

OIDN (Open Image Denoise)

Unlike JNLM, OIDN uses a machine learning approach to denoising lightmaps. It features a model specifically trained to remove noise from lightmaps while preserving more shadow detail in most scenes compared to JNLM.

OIDN can run on the GPU if hardware acceleration is configured. With a modern high-end GPU, this can provide a speedup of over 50× over CPU-based denoising:

- On AMD GPUs, HIP must be installed and configured.
- On NVIDIA GPUs, CUDA must be installed and configured. This may automatically be done by the NVIDIA installer, but on Linux, CUDA libraries may not be installed by default. Double-check that the CUDA packages from your Linux distribution are installed.
- On Intel GPUs, SYCL must be installed and configured.

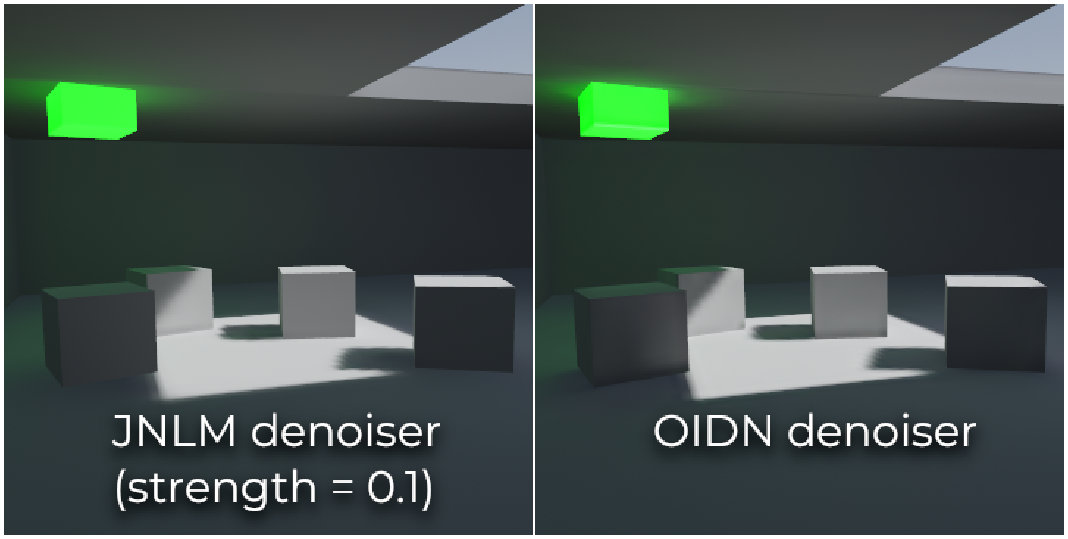
If hardware acceleration is not available, OIDN will fall back to multithreaded CPU-based denoising. To confirm whether GPU-based denoising is working, use a GPU utilization monitor while baking lightmaps and look at the GPU utilization percentage and VRAM utilization while the denoising step is shown in the Godot editor. The `nvidia-smi` command line tool can be useful for this.

OIDN is not included with Godot due to its relatively large download size. You can download precompiled OIDN binary packages from its website [https://www.openimagedenoise.org/downloads.html]. Extract the package to a location on your PC, then specify the path to the `oidnDenoise` executable in the Editor Settings (**FileSystem > Tools > OIDN > OIDN Denoise Path**). This executable is located within the `bin` folder of the binary package you extracted.

After specifying the path to the OIDN denoising executable, change the denoising method in the project settings by setting **Rendering > Lightmapping > Denoiser** to **OIDN**. This will affect all lightmap bakes on this project after the setting is changed.

Note

The denoising method is configured in the project settings instead of the editor settings. This is done so that different team members working on the same project are assured to be using the same denoising method for consistent results.



Comparison between JNLM and OIDN denoisers. Notice how OIDN better preserves detail and reduces seams across different objects.

Dynamic objects

Unlike VoxelGI and SDFGI, dynamic objects receive indirect lighting differently compared to static objects. This is because lightmapping is only performed on static objects.

To display indirect lighting on dynamic objects, a 3D probe system is used, with light probes being spread throughout the scene. When baking lightmaps, the lightmapper will calculate the amount of *indirect* light received by the probe. Direct light is not stored within light probes, even for lights that have their bake mode set to **Static** (as dynamic objects continue to be lit in real-time).

There are 2 ways to add light probes to a scene:

- **Automatic:** Set **Gen Probes > Subdiv** to a value other than **Disabled**, then bake lightmaps. The default is `8`, but you can choose a greater value to improve precision at the cost of longer bake times and larger output file size.

- **Manual:** In addition or as an alternative to generating probes automatically, you can add light probes manually by adding LightmapProbe nodes to the scene. This can be used to improve lighting detail in areas frequently travelled by dynamic objects. After placing LightmapProbe nodes in the scene, you must bake lightmaps again for them to be effective.

Note

After baking lightmaps, you will notice white spheres in the 3D scene that represent how baked lighting will affect dynamic objects. These spheres do **not** appear in the running project.

If you want to hide these spheres in the editor, toggle **View > Gizmos > LightmapGI** at the top of the 3D editor (a "closed eye" icon indicates the gizmo is hidden).

Lightmap data

The **Data > Light Data** property in the LightmapGI node contains the lightmap data after baking. Textures are saved to disk, but this also contains the capture data for dynamic objects, which can be heavy. If you are using a scene in `.tscn` format, you should save this resource to an external binary `.lmbake` file to avoid bloating the `.tscn` scene with binary data encoded in Base64.

Tip

The generated EXR file can be viewed and even edited using an image editor to perform post-processing if needed. However, keep in mind that changes to the EXR file will be lost when baking lightmaps again.

Reducing LightmapGI artifacts

If you notice LightmapGI nodes popping in and out of existence as the camera moves, this is most likely because the engine is rendering too many LightmapGI instances at once. Godot is limited to rendering 8 LightmapGI nodes at once, which means up to 8 instances can be in the camera view before some of them will start flickering.

Reflection probes

Note

Reflection probes are only supported in the Clustered Forward and Forward Mobile rendering backends, not the Compatibility backend.

As stated in the Standard Material 3D and ORM Material 3D, objects can show reflected and/or diffuse light. Reflection probes are used as a source of reflected *and* ambient light for objects inside their area of influence. They can be used to provide more accurate reflections than VoxelGI and SDFGI while being fairly cheap on system resources.

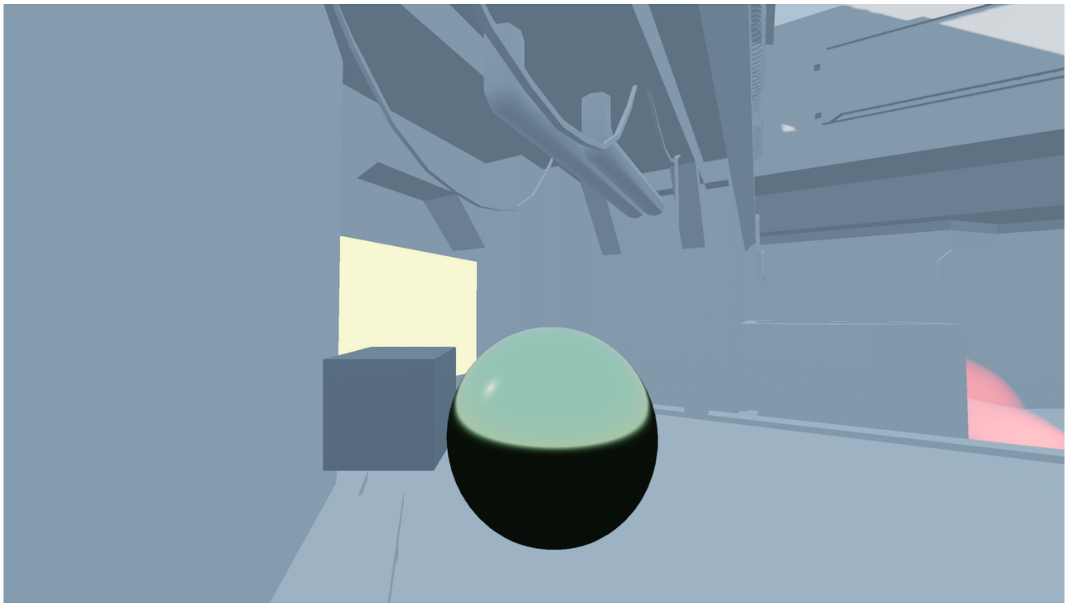
Since reflection probes can also store ambient light, they can be used as a low-end alternative to VoxelGI and SDFGI when baked lightmaps aren't viable (e.g. in procedurally generated levels).

Reflection probes can also be used at the same time as screen-space reflections to provide reflections for off-screen objects. In this case, Godot will blend together the screen-space reflections and reflections from reflection probes.

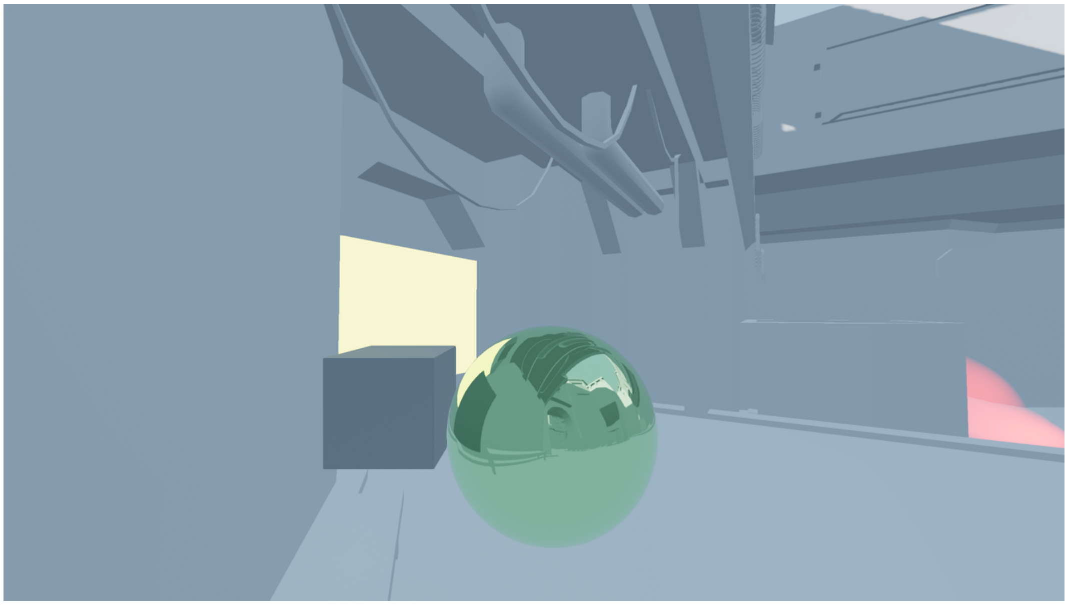
See also

Not sure if ReflectionProbe is suited to your needs? See Which global illumination technique should I use? for a comparison of GI techniques available in Godot 4.

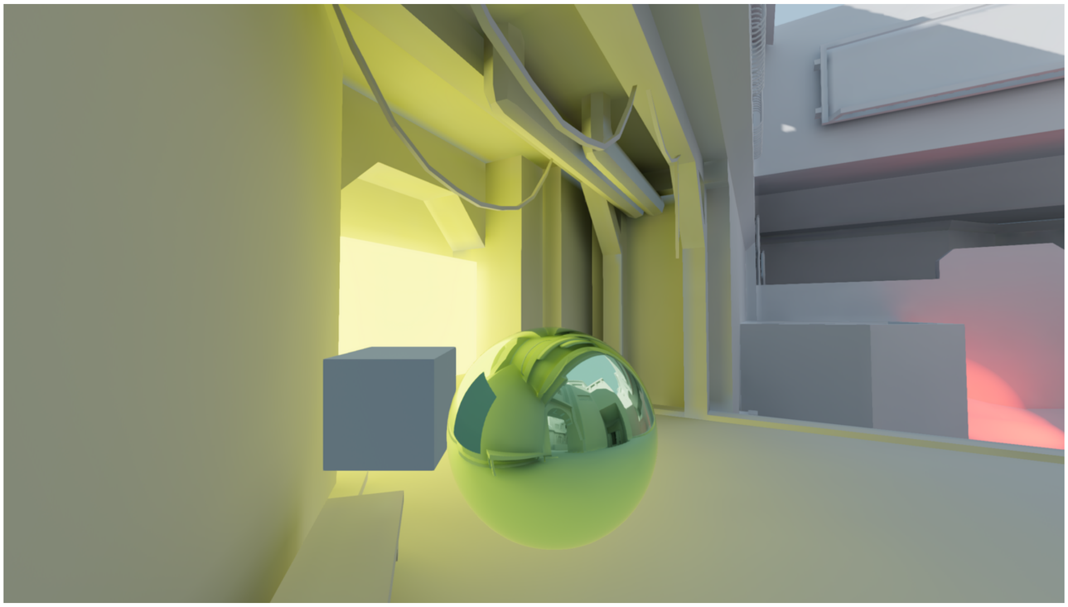
Visual comparison



Reflection probe disabled. Environment sky is used as a fallback.

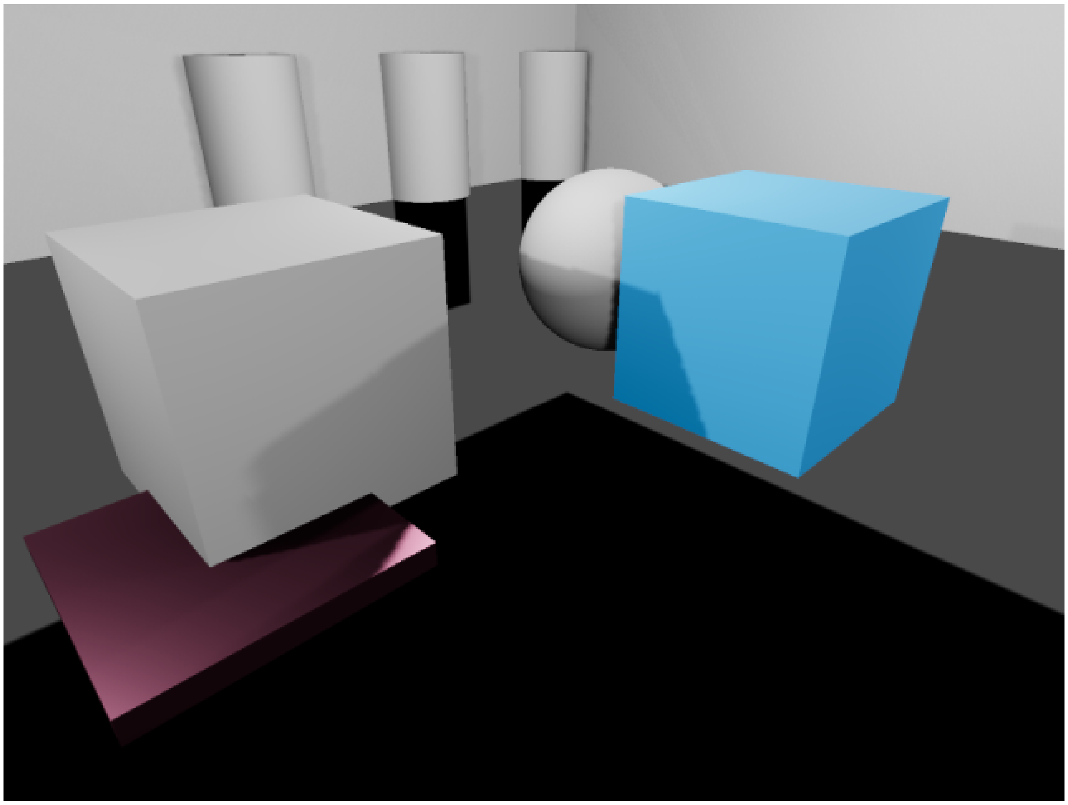


Reflection probe enabled.

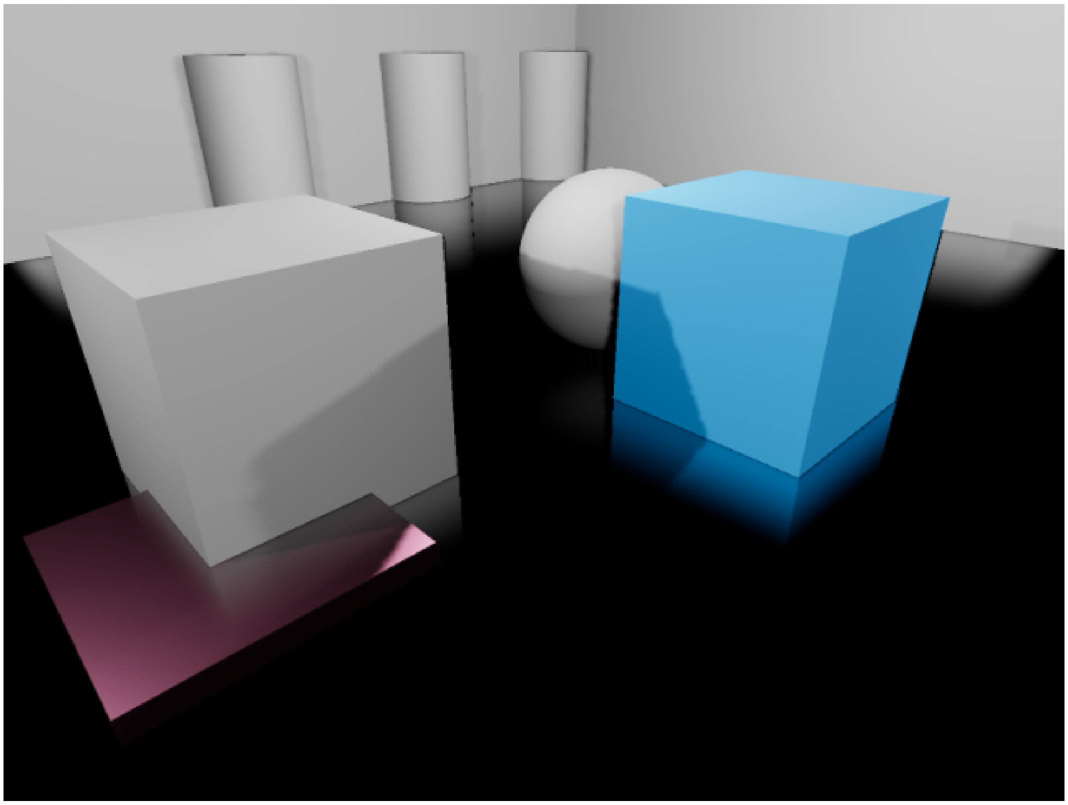


Reflection probe enabled with LightmapGI used at the same time. The lightmap appears in the reflection.

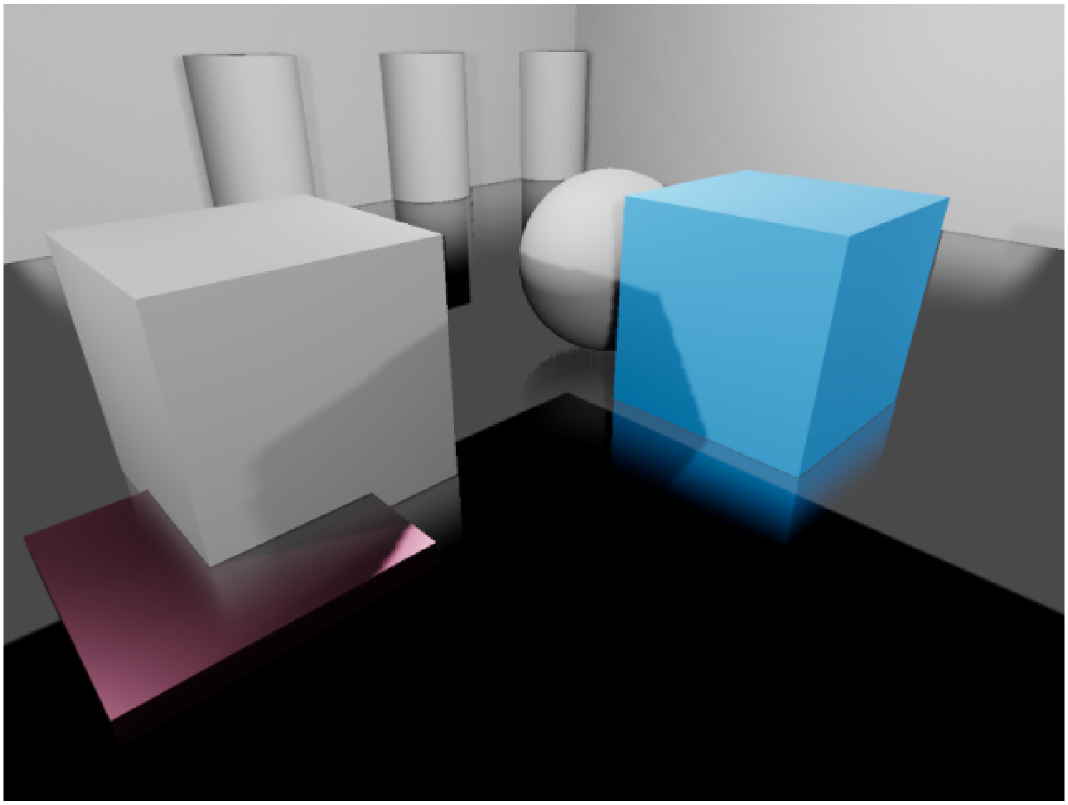
By combining reflection probes with screen-space reflections, you can get the best of both worlds: high-quality reflections for general room structure (that remain present when off-screen), while also having real-time reflections for small details.



Reflections in a room using ReflectionProbe only. Notice how small details don't have any reflections.



Reflections in a room using screen-space reflections only. Notice how the reflection on the sides of the room's walls is partly missing due to being off-screen.



Reflections in a room using ReflectionProbe and screen-space reflections together. The screen-space reflections are blended with the reflection probe, acting as a fallback in situations where the reflection probe fails to display any reflection.

Setting up a ReflectionProbe

- Add a ReflectionProbe node.
- Configure the ReflectionProbe's extents in the inspector to fit your scene. To get reasonably accurate reflections, you should generally have one ReflectionProbe node per room (sometimes more for large rooms).

Tip

Remember that ReflectionProbe extents don't have to be square, and you can even rotate the ReflectionProbe node to fit rooms that aren't aligned with the X/Z grid. Use this to your advantage to better cover rooms without having to place too many ReflectionProbe nodes.

ReflectionProbe properties

- **Update Mode:** Controls when the reflection probe updates. **Once** only renders the scene once every time the ReflectionProbe is moved. This makes it much faster to render compared to the **Always** update mode, which forces the probe to re-render everything around it every frame. Leave this property on **Once** (default) unless you need the reflection probe to update every frame.
- **Intensity:** The brightness of the reflections and ambient lighting. This usually doesn't need to be changed from its default value of `1.0`, but you can decrease it `1.0` if you find that reflections look too strong.
- **Max Distance:** Controls the maximum distance used by the ReflectionProbe's internal camera. The distance is always at least equal to the **Extents**, but this can be increased to make objects located outside the extents visible in reflections. *This property does not affect the maximum distance at which the ReflectionProbe itself is visible.*
- **Extents:** The area that will be affected by the ReflectionProbe's lighting and reflections.
- **Origin Offset:** The origin to use for the internal camera used for reflection probe rendering. This must always be constrained within the **Extents**. If needed, adjust this to prevent the reflection from being obstructed by a solid object located exactly at the center of the ReflectionProbe.

- **Box Projection:** Controls whether parallax correction should be used when rendering the reflection probe. This adjusts the reflection's appearance depending on the camera's position (relative to the reflection probe). This has a small performance cost, but the quality increase is often worth it in box-shaped rooms. Note that this effect doesn't work quite as well in rooms with less regular shapes (such as ellipse-shaped rooms).
- **Interior:** If enabled, ambient lighting will not be sourced from the environment sky, and the background sky won't be rendered onto the reflection probe.
- **Enable Shadows:** Controls whether real-time light shadows should be rendered within the reflection probe. Enable this to improve reflection quality at the cost of performance. This should be left disabled for reflection probes with the **Always** mode, as it's very expensive to render reflections with shadows every frame. Fully baked light shadows are not affected by this setting and will be rendered in the reflection probe regardless.
- **Cull Mask:** Controls which objects are visible in the reflection. This can be used to improve performance by excluding small objects from the reflection. This can also be used to prevent an object from having self-reflection artifacts in situations where **Origin Offset** can't be used.
- **Mesh LOD Threshold:** The automatic level of detail threshold to use for rendering meshes within the reflection. This only affects meshes that have automatic LODs generated for them. Higher values can improve performance by using less detailed geometry, especially for objects that are far away from the reflection's origin. The visual difference of using less detailed objects is usually not very noticeable during gameplay, especially in rough reflections.

The Ambient category features several properties to adjust ambient lighting rendered by the ReflectionProbe:

- **Mode:** If set to **Disabled**, no ambient light is added by the probe. If set to **Environment**, the ambient light color is

automatically sampled from the environment sky (if **Interior** is disabled) and the reflection's average color. If set to **Constant Color**, the color specified in the **Color** property is used instead. The **Constant Color** mode can be used as an approximation of area lighting.
- **Color:** The color to use when the ambient light mode is set to **Constant Mode**.
- **Color Energy:** The multiplier to use for the ambient light custom **Color**. This only has an effect when the ambient light mode is **Custom Color**.

ReflectionProbe blending

To make transitions between reflection sources smoother, Godot supports automatic probe blending:

- Up to 4 ReflectionProbes can be blended together at a given location. A ReflectionProbe will also fade out smoothly back to environment lighting when it isn't touching any other ReflectionProbe node.
- SDFGI and VoxelGI will blend in smoothly with ReflectionProbes if used. This allows placing ReflectionProbes strategically to get more accurate (or fully real-time) reflections where needed, while still having rough reflections available in the VoxelGI or SDFGI's area of influence.

To make several ReflectionProbes blend with each other, you need to have part of each ReflectionProbe overlap each other's area. The extents should only overlap as little possible with other reflection probes to improve rendering performance (typically a few units in 3D space).

Limitations

When using the Forward+ renderer, Godot uses a *clustering* approach for reflection probe rendering. As many reflection probes as desired can be added (as long as performance allows). However, there's still a default limit of 512 *clustered elements* that can be present in the current camera view. A clustered element is an omni light, a spot light, a decal or a reflection probe. This limit can be increased by adjusting the **Rendering > Limits > Cluster Builder > Max Clustered Elements** advanced project setting.

When using the Forward Mobile backend, only 8 reflection probes can be applied on each individual Mesh *resource*. If there are more reflection probes affecting a single mesh, not all of them will be rendered on the mesh.

Similarly, when using the Compatibility backend, up to 2 reflection probes can be applied per mesh. If more than 2 reflection probes affect a single mesh, additional probes will not be rendered.

Faking global illumination

Why fake global illumination?

Godot provides several global illumination (GI) techniques, all with their advantages and drawbacks. Nonetheless, it remains possible to avoid using any GI technique and use a handmade approach instead. There are a few reasons for using a "handmade" approach to global illumination instead of VoxelGI, SDFGI or baked lightmaps:

- You need to have good rendering performance, but can't afford going through a potentially cumbersome lightmap baking process.
- You need an approach to GI that is fully real-time *and* works in procedurally generated levels.
- You need an approach to GI that is fully real-time *and* does not suffer from significant light leaks.

The approaches described below only cover indirect diffuse lighting, not specular lighting. For specular lighting, consider using ReflectionProbes which are usually cheap enough to be used in conjunction with this fake GI approach.

See also

Not sure if faking global illumination with lights is suited to your needs? See Which global illumination technique should I use? for a comparison of GI techniques available in Godot 4.

Faking DirectionalLight3D global illumination

While the sky provides its own directional lighting, the scene's main DirectionalLight3D node typically emits a large amount of light. When using a GI technique, this light would be reflected on solid surfaces and would bounce back on most outdoors shaded surfaces.

We can fake this by adding a second DirectionalLight3D node with the following changes:

- Rotate the light by 180 degrees. This allows it to represent lighting bounced by the main DirectionalLight3D node.
- Set **Shadows** to **Off**. This reduces the secondary light's performance burden while also allowing shaded areas to receive *some* lighting (which is what we want here).
- Set **Energy** to 10-40% of the original value. There is no "perfect" value, so experiment with various energy values depending on the light and your typical material colors.
- Set **Specular** to `0.0`. Indirect lighting shouldn't emit visible specular lobes, so we need to disable specular lighting entirely for the secondary light.

Note

This approach works best in scenes that are mostly outdoors. When going indoors, the secondary DirectionalLight3D's light will still be visible as this light has shadows disabled.

This can be worked around by smoothly decreasing the secondary DirectionalLight3D's energy when entering an indoor area (and doing the opposite when leaving the indoor area). For instance, this can be achieved using an Area3D node and AnimationPlayer.

Faking positional light global illumination

It's possible to follow the same approach as DirectionalLight3D for positional lights (OmniLight3D and SpotLight3D). However, this will require more manual work as this operation needs to be repeated for every positional light node in the scene to look good.

In an ideal scenario, additional OmniLight3Ds should be added at every location where a significant amount of light hits a bright enough surface. However, due to time constraints, this isn't always easily feasible (especially when performing procedural level generation).

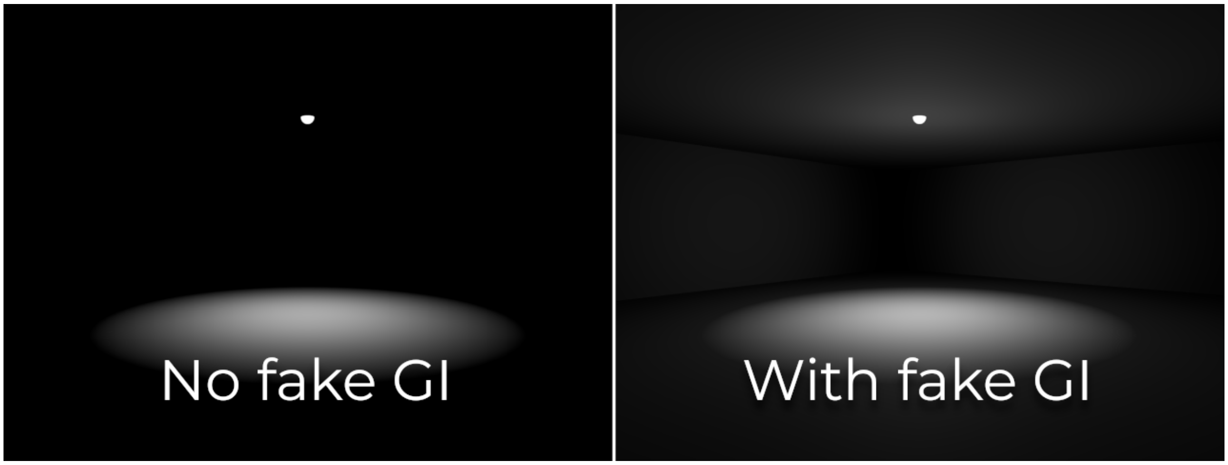
If you're in a hurry, you can place a secondary OmniLight3D node at the same position as the main OmniLight3D node. You can add this node as a child of the main OmniLight3D node to make it easy to move and hide both nodes at the same time.

In the secondary OmniLight3D node, perform the following changes:

- Increase the light's **Range** by 25-50%. This allows the secondary light to lighten what was previously not lit by the original light.
- Set **Shadows** to **Off**. This reduces the secondary light's performance burden while also allowing shaded areas to receive *some* lighting (which is what we want here).
- Set **Energy** to 10-40% of the original value. There is no "perfect" value, so experiment with various energy values depending on the light and its surroundings.
- Set **Specular** to 0. Indirect lighting shouldn't emit visible specular lobes, so we need to disable specular lighting entirely for the secondary light.

For SpotLight3D, the same trick can be used. In this case, the secondary OmniLight3D should be placed in a way that reflects where *most* light will be bounced. This is usually close to the SpotLight3D's primary impact location.

In the example below, a SpotLight3D node is used to light up the room's floor. However, since there is no indirect lighting, the rest of the room remains entirely dark. In real life, the room's walls and ceiling would be lit up by light bouncing around. Using an OmniLight3D node positioned between the SpotLight3D's origin and the floor allows simulating this effect:



Environment and post-processing

Godot 4 provides a redesigned Environment resource, as well as a new post-processing system with many available effects right out of the box.

Note

As of Godot 4, Environment *performance/quality* settings are defined in the project settings instead of in the Environment resource. This makes global adjustments easier, as you no longer have to tweak Environment resources individually to suit various hardware configurations.

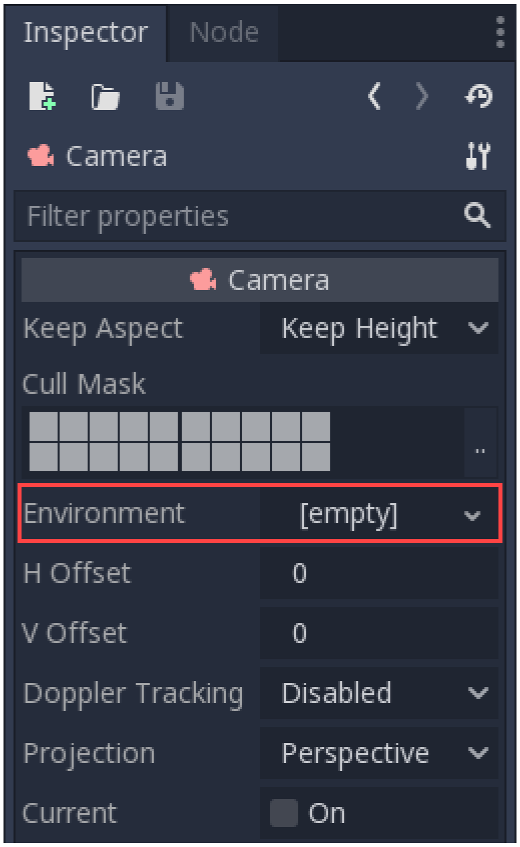
Note that most Environment performance/quality settings are only visible after enabling the **Advanced** toggle in the Project Settings.

Environment

The Environment resource stores all the information required for controlling the 2D and 3D rendering environment. This includes the sky, ambient lighting, tone mapping, effects, and adjustments. By itself, it does nothing, but you can enable it by using it in one of the following locations, in order of priority:

Camera3D node (high priority)

An Environment can be set to a Camera3D node. It will have priority over any other setting.



Camera

Camera

Keep Aspect: Keep Height

Cull Mask

Environment: [empty]

H Offset: 0

V Offset: 0

Doppler Tracking: Disabled

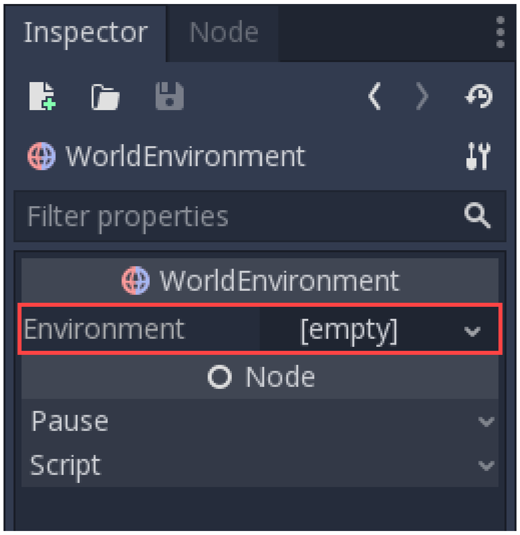
Projection: Perspective

Current: ☐ On

This is mostly useful when you want to override an existing environment, but in general it's a better idea to use the option below.

WorldEnvironment node (medium priority, recommended)

The WorldEnvironment node can be added to any scene, but only one can exist per active scene tree. Adding more than one will result in a warning.


Inspector
Node
WorldEnvironment
Filter properties
WorldEnvironment
Environment
[empty]
Node
Pause
Script

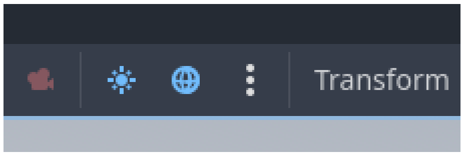
Any Environment added has higher priority than the default Environment (explained below). This means it can be overridden on a per-scene basis, which makes it quite useful.

Preview environment and sun (low priority)

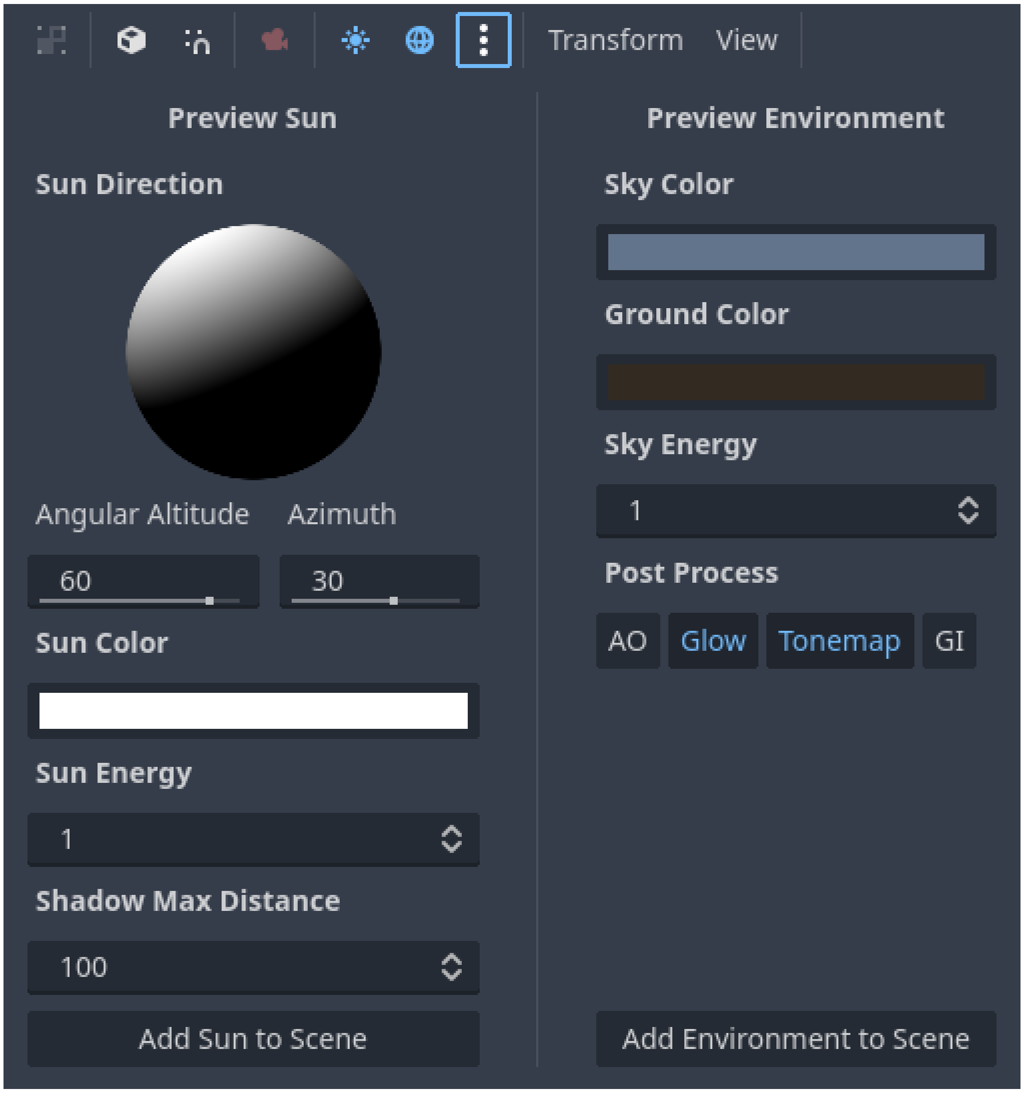
Note

Since Godot 4, the preview environment and sun system replace the `default_env.tres` file that was used in Godot 3 projects.

If no WorldEnvironment node or DirectionalLight3D node is present in the current scene, the editor will display a preview environment and sun instead. This can be disabled using the buttons at the top of the 3D editor:



Clicking on the 3 vertical dots on the right will display a dialog which allows you to customize the appearance of the preview environment:


Transform
View
Preview Sun
Sun Direction
Angular Altitude
Azimuth
Sun Color
Sun Energy
Shadow Max Distance
Add Sun to Scene
Preview Environment
Sky Color
Ground Color
Sky Energy
Post Process
AO
Glow
Tonemap
GI
Add Environment to Scene

Angular Altitude: 60

Azimuth: 30

Sun Energy: 1

Shadow Max Distance: 100

Sky Energy: 1

The preview sun and sky is only visible in the editor, not in the running project. Using the buttons at the bottom of the dialog, you can add the preview sun and sky into the scene as nodes.

Tip

If you hold `Shift` while clicking **Add Sun to Scene** or **Add Environment to Scene** in the preview environment editor, this will add both a preview sun and environment to the current scene (as if you clicked both buttons separately). Use this to speed up project setup and prototyping.

Camera attributes

Note

In Godot 4, exposure and depth of field information was split from the Environment resource into a separate CameraAttributes resource. This allows adjusting those properties independently of other Environment settings more easily.

The CameraAttributes resource stores exposure and depth of field information. It also allows enabling automatic exposure adjustments depending on scene brightness.

There are two kinds of CameraAttribute resources available:

- **CameraAttributesPractical:** Features are exposed using arbitrary units, which are easier to reason about for most game use cases.
- **CameraAttributesPhysical:** Features are exposed using real world units, similar to a digital camera. For example, field of view is set using a focal length in millimeters instead of a value in degrees. Recommended when physical accuracy is important, such as for photorealistic rendering.

Both CameraAttribute resource types allow you to use the same features, but they are configured differently. If you don't know which one to choose, use **CameraAttributesPractical**.

Note

Using a CameraAttributesPhysical on a Camera3D node will lock out FOV and aspect adjustments in that Camera3D, as field of view is adjusted in the CameraAttributesPhysical resource instead. If used in a WorldEnvironment, the CameraAttributesPhysical will not override any Camera3D in the scene.

A CameraAttributes resource can be added to a Camera3D or a WorldEnvironment node. When the current camera has a CameraAttributes set, it will *override* the one set in WorldEnvironment (if any).

In most situations, setting the CameraAttributes resource on the Camera3D node instead of the WorldEnvironment is recommended. Unlike WorldEnvironment, assigning the CameraAttributes resource to the Camera3D node prevents depth of field from displaying in the 3D editor viewport, unless the camera is being previewed.

Environment options

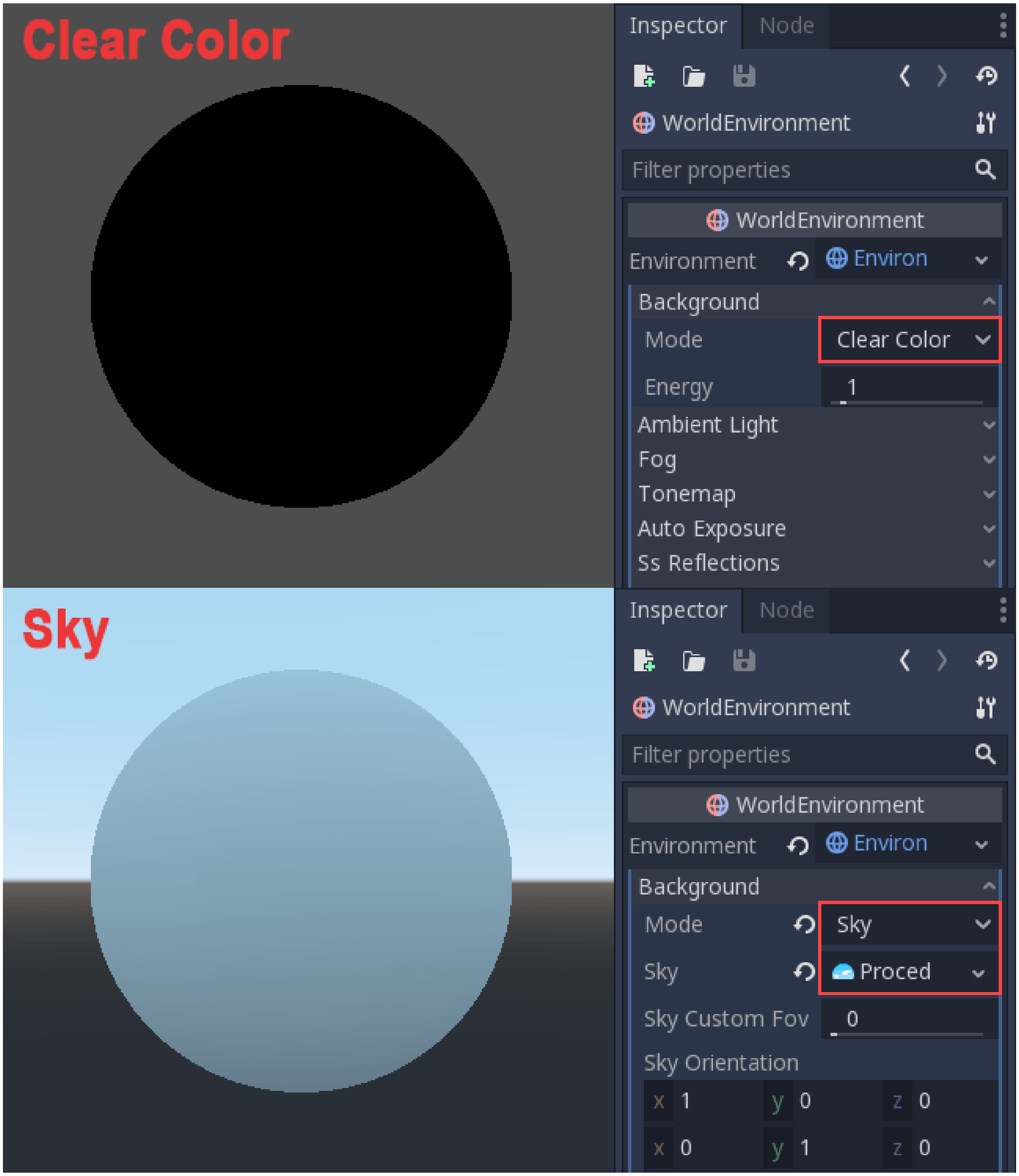
The following is a detailed description of all environment options and how they are intended to be used.

Background

The Background section contains settings on how to fill the background (parts of the screen where objects were not drawn). The background not only serves the purpose of displaying an image or color. By default, it also affects how objects are affected by ambient and reflected light. This is called image-based lighting (IBL).

As a result, the background sky may greatly impact your scene's overall appearance, even if the sky is never directly visible on screen.

This should be taken into account when tweaking lighting in your scene.


Clear Color
Inspector
Node
WorldEnvironment
Filter properties
WorldEnvironment
Sky
Inspector
Node
WorldEnvironment
Filter properties
WorldEnvironment

Environment: Environ
Background
Mode: Clear Color
Energy: 1
Ambient Light
Fog
Tonemap
Auto Exposure
Ss Reflections

Environment: Environ
Background
Mode: Sky
Sky: Proced
Sky Custom Fov: 0
Sky Orientation
x 1 y 0 z 0
x 0 y 1 z 0

There are several background modes available:

- **Clear Color** uses the default clear color defined in the project settings. The background will be a constant color.
- **Custom Color** is like Clear Color, but with a custom color value.
- **Sky** lets you define a background sky material (see below). By default, objects in the scene will reflect this sky material and absorb ambient light from it.
- **Canvas** displays the 2D scene as a background to the 3D scene. This can be used to make environment effects visible on 2D rendering, such as glow in 2D.
- **Keep** does not draw any sky, keeping what was present on previous frames instead. This improves performance in purely indoor scenes, but creates a "hall of mirrors" visual glitch if the sky is visible at any time.

Sky materials

When using the **Sky** background mode (or the ambient/reflected light mode is set to **Sky**), a Sky subresource becomes available to edit in the Environment resource. Editing this subresource allows you to create a SkyMaterial resource within the Sky.

There are 3 built-in sky materials to choose from:

- **PanoramaSkyMaterial:** Use a 360 degree panorama sky image (2:1 aspect ratio recommended). To benefit from high dynamic range, the panorama image must be in an HDR-compatible format such as `.hdr` or `.exr` rather than a standard dynamic range format like `.png` or `.jpg`.
- **ProceduralSkyMaterial:** Use a procedurally generated sky with adjustable ground, sun, sky and horizon colors. This is the type of sky used in the editor preview. The sun's position is automatically derived from the first 4 DirectionalLight3D nodes present in the scene. There can be up to 4 suns at a given time.
- **PhysicalSkyMaterial:** Use a physically-based procedural sky with adjustable scattering parameters. The sun's position is

automatically derived from the first DirectionalLight3D node present in the scene. PhysicalSkyMaterial is slightly more expensive to render compared to ProceduralSkyMaterial. There can be up to 1 sun at a given time.

Panorama sky images are sometimes called HDRIs (High Dynamic Range Images). You can find freely licensed HDRIs on Poly Haven [https://polyhaven.com/hdris].

Note

HDR PanoramaSkyMaterial textures with very bright spots (such as real life photos with the sun visible) may result in visible sparkles on ambient and specular reflections. This is caused by the texture's peak exposure being too high.

To resolve this, select the panorama texture in the FileSystem dock, go to the Import dock, enable **HDR Clamp Exposure** then click **Reimport**.

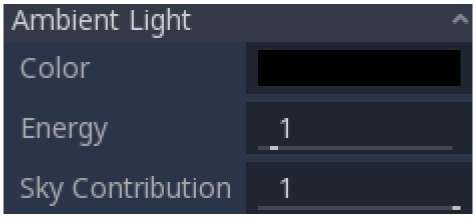
If you need a custom sky material (e.g. for procedural clouds), you can create a custom sky shader.

Ambient light

Ambient light (as defined here) is a type of light that affects every piece of geometry with the same intensity. It is global and independent of lights that might be added to the scene. Ambient light is one of the two components of image-based lighting. Unlike reflected light, ambient light does not vary depending on the camera's position and viewing angle.

There are several types of ambient light to choose from:

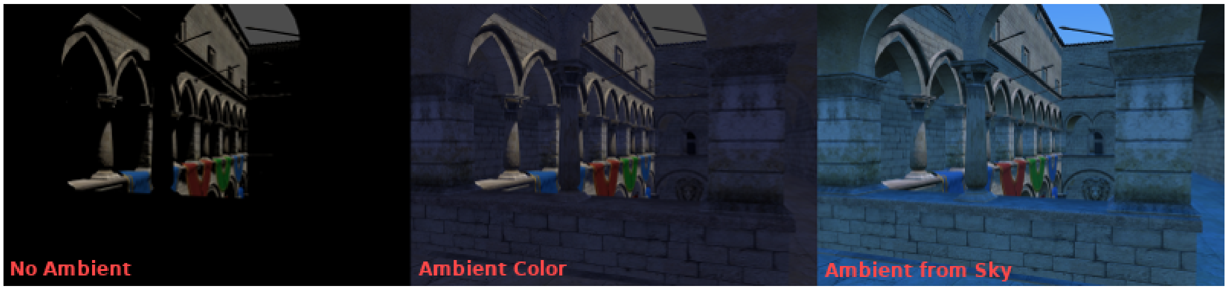
- **Background:** Source ambient light from the background, such as the sky, custom color or clear color (default). Ambient light intensity will vary depending on the sky image's contents, which can result in more visually appealing ambient lighting. A sky must be set as background for this mode to be visible.
- **Disabled:** Do not use any ambient light. Useful for purely indoor scenes.
- **Color:** Use a constant color for ambient light, ignoring the background sky. Ambient light intensity will be the same on all sides, which may result in the scene's lighting looking more flat. Useful for indoor scenes where pitch black shadows may be too dark, or to maximize performance on low-end devices.
- **Sky:** Source ambient light from a specified sky, even if the background is set to a mode other than **Sky**. If the background mode is already **Sky**, this mode behaves identically to **Background**.



Color	■
Energy	1
Sky Contribution	1

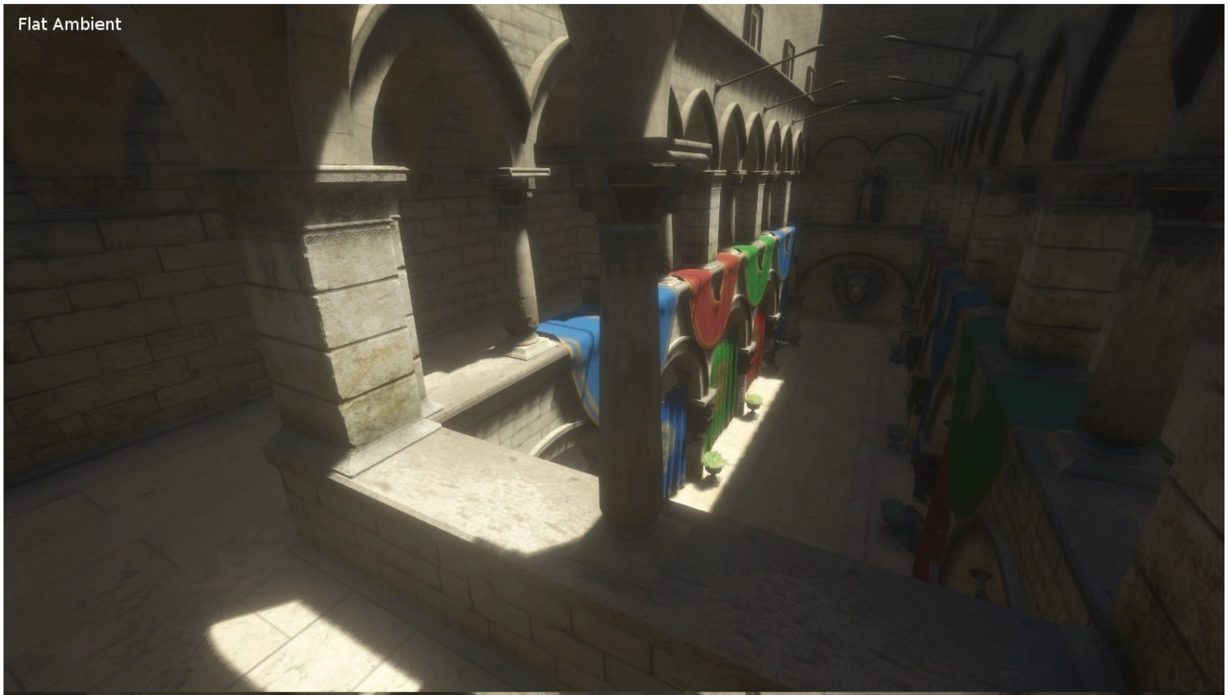
When the ambient light mode is set to Sky or Background (and background is set to Sky), it's possible to blend between the ambient color and sky using the **Sky Contribution** property. This value is set to `1.0` by default, which means that only the ambient sky is used. The ambient color is ignored unless **Sky Contribution** is decreased below `1.0`.

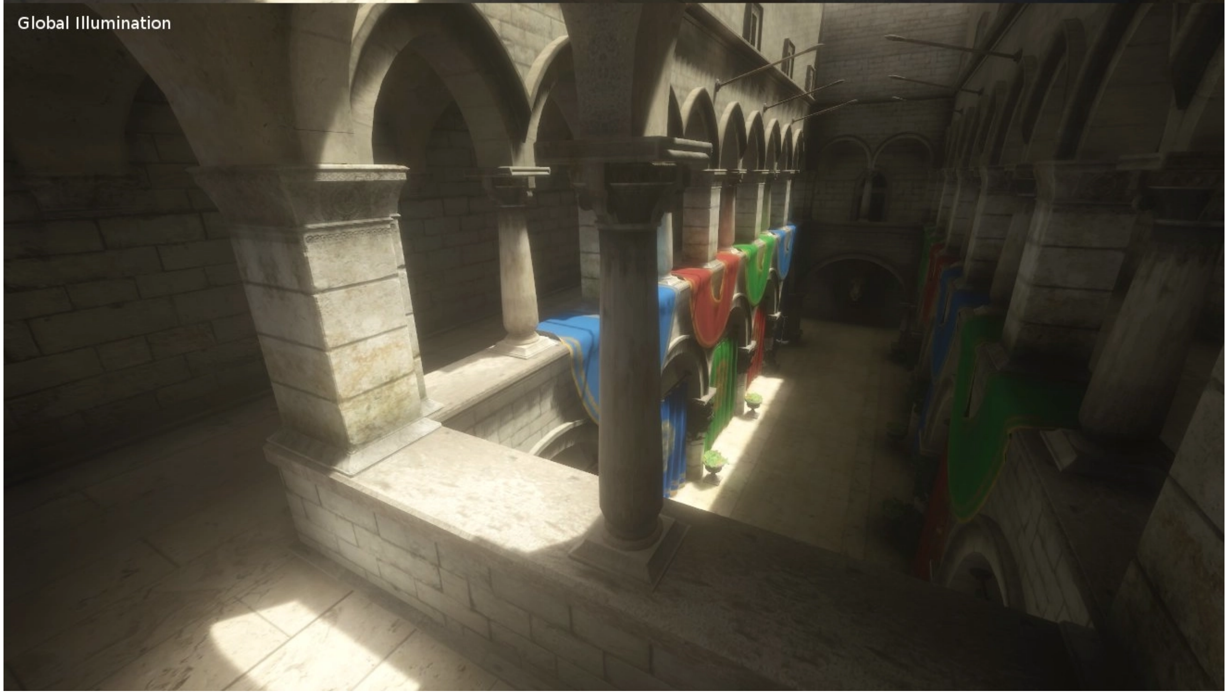
Here is a comparison of how different ambient light affects a scene:



Finally, there is an **Energy** setting which is a multiplier. It's useful when working with HDR.

In general, you should only rely on ambient light alone for simple scenes or large exteriors. You may also do so to boost performance. Ambient light is fast to render, but it doesn't provide the best lighting quality. It's better to generate ambient light from ReflectionProbe, VoxelGI or SDFGI, as these will simulate how indirect light propagates more accurately. Below is a comparison, in terms of quality, between using a flat ambient color and a VoxelGI:


Flat Ambient


Global Illumination

Using one of the methods described above will replace constant ambient lighting with ambient lighting from the probes.

Reflected light

Reflected light (also called specular light) is the other of the two components of image-based lighting.

Reflected light can be set to one of 3 modes:

- **Background:** Reflect from the background, such as the sky, custom color or clear color (default).
- **Disabled:** Do not reflect any light from the environment. Useful for purely indoor scenes, or to maximize performance on low-end devices.
- **Sky:** Reflect from the background sky, even if the background is set to a mode other than **Sky**. If the background mode is already **Sky**, this mode behaves identically to **Background**.

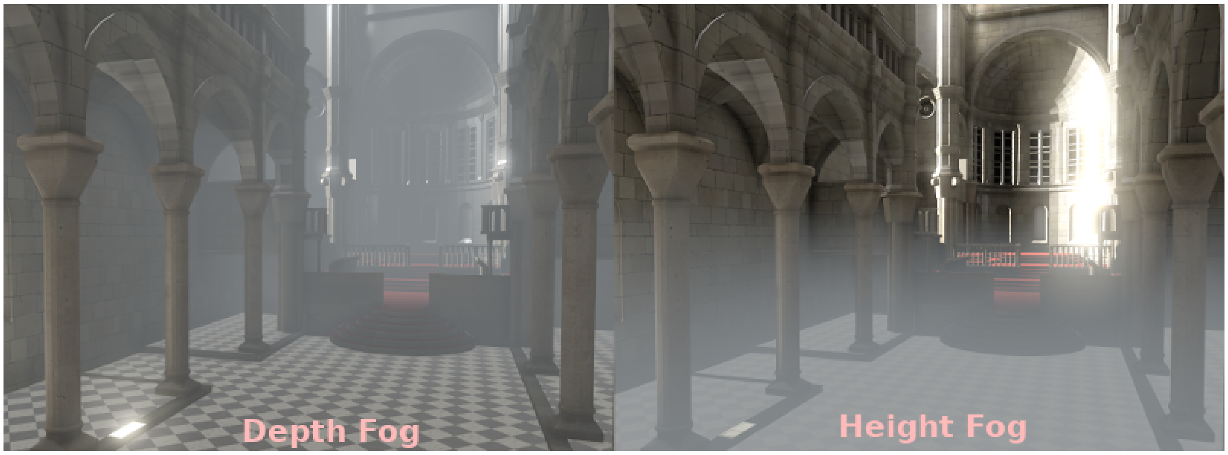
Fog

Note

This section refers to non-volumetric fog only. It is possible to use both non-volumetric fog and Volumetric fog and fog volumes at the same time.

Fog, as in real life, makes distant objects fade away into a uniform color. There are two kinds of fog in Godot:

- **Depth Fog:** This one is applied based on the distance from the camera.
- **Height Fog:** This one is applied to any objects below (or above) a certain height, regardless of the distance from the camera.

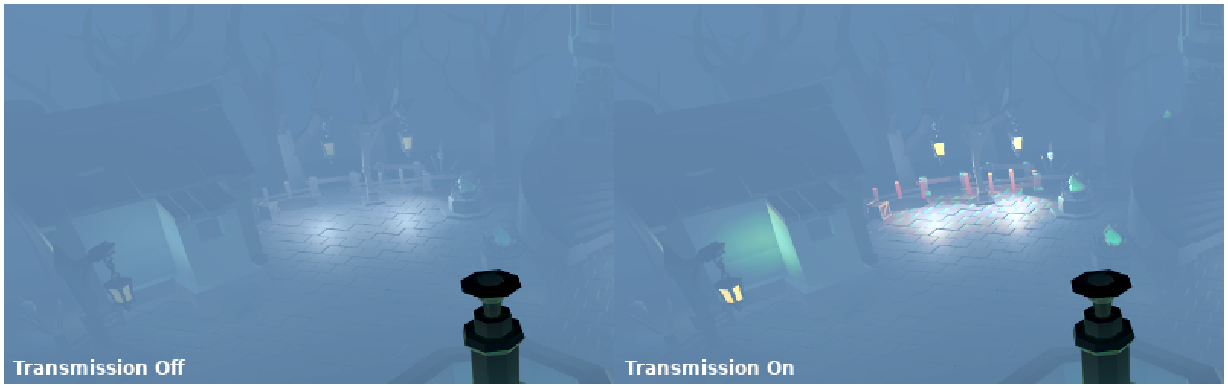


Both of these fog types can have their curve tweaked, making their transition more or less sharp.

Two properties can be tweaked to make the fog effect more interesting:

The first is **Sun Amount**, which makes use of the Sun Color property of the fog. When looking towards a directional light (usually a sun), the color of the fog will be changed, simulating the sunlight passing through the fog.

The second is **Transmit Enabled** which simulates more realistic light transmittance. In practice, it makes light stand out more across the fog.



Note

Fog can cause banding to appear on the viewport, especially at higher density levels. See Color banding for guidance on reducing banding.

Volumetric Fog

Volumetric fog provides a realistic fog effect to the scene, with fog color being affected by the lights that traverse the fog.

See also

See Volumetric fog and fog volumes for documentation on setting up volumetric fog.

Tonemap

Tonemap selects the tonemapping curve that will be applied to the scene, from a list of standard curves used in the film and game industries. Tonemapping operators other than Linear are used to make light and dark areas more homogeneous, while also avoiding clipping of bright highlights.

The tone mapping options are:

- **Mode:** The tone mapping mode to use.
    - **Linear:** The default tonemapping mode. This is the fastest and simplest tonemapping operator, but it causes bright lighting to look blown out, with noticeable clipping in the output colors.
    - **Reinhardt:** Performs a variation on rendered pixels' colors by this formula: `color = color / (1 + color)`. This avoids clipping bright highlights, but the resulting image can look a bit dull.

  - **Filmic:** This avoids clipping bright highlights, with a resulting image that usually looks more vivid than Reinhardt.
  - **ACES:** Academy Color Encoding System tonemapper. ACES is slightly more expensive than other options, but it handles bright lighting in a more realistic fashion by desaturating it as it becomes brighter. ACES typically has a more contrasted output compared to Reinhardt and Filmic. ACES is the recommended option when aiming for photorealistic visuals. This tonemapping mode was called "ACES Fitted" in Godot 3.x.
- **Exposure:** Tone mapping exposure which simulates amount of light received over time (default: `1.0`). Higher values result in an overall brighter appearance. If the scene appears too dark as a result of a tonemapping operator or whitepoint change, try increasing this value slightly.
- **White:** Tone mapping whitepoint, which simulates where in the scale white is located (default: `1.0`). For photorealistic lighting, recommended values are between `6.0` and `8.0`. Higher values result in less blown out highlights, but make the scene appear slightly darker as a whole.

Mid- and post-processing effects

The Environment resource supports many popular mid- and post-processing effects.

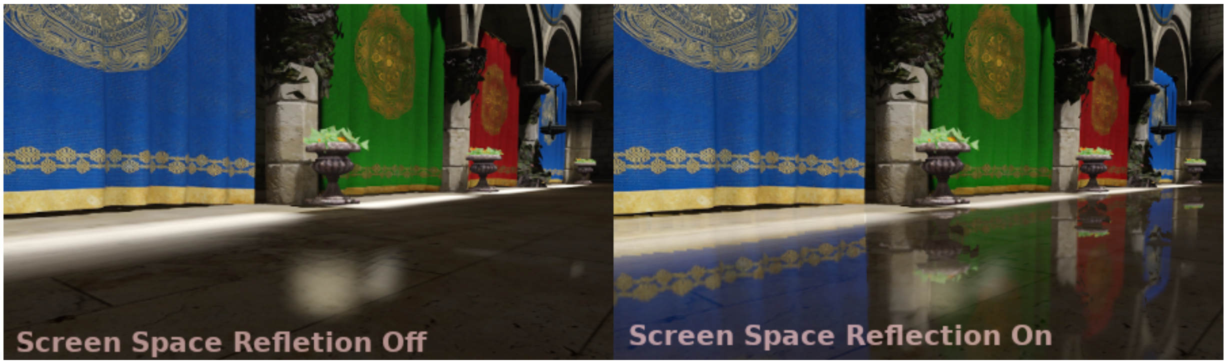
Note

Screen-space effects such as SSR, SSAO, SSIL and glow do not operate on geometry that is located outside the camera view or is occluded by other opaque geometry. Consider this when tweaking their settings to avoid distracting changes during gameplay.

Screen-Space Reflections (SSR)

This feature is only available when using the Forward+ backend, not Mobile or Compatibility.

While Godot supports several sources of reflection data such as Reflection probes, they may not provide enough detail for all situations. Scenarios where screen-space reflections make the most sense are when objects are in contact with each other (object over floor, over a table, floating on water, etc).



On top of providing more detail, screen-space reflections also work in real-time (while other types of reflections are usually precomputed). This can be used to make characters, cars, etc. reflect on surrounding surfaces when moving around.

Screen-space reflections can be used at the same time as other reflection sources to benefit from detailed reflections when possible, while having a fallback when screen-space reflections cannot be used (for example, to reflect off-screen objects).

A few user-controlled parameters are available to better tweak the technique:

- **Max Steps:** Determines the length of the reflection. The bigger this number, the more costly it is to compute.
- **Fade In:** Allows adjusting the fade-in curve, which is useful to make the contact area softer.

- **Fade Out:** Allows adjusting the fade-out curve, so the step limit fades out softly.
- **Depth Tolerance:** Can be used to allow screen-space rays to pass behind objects. The rays will treat each object as if it has this depth in determining if it can pass behind the object. Higher values will make screen-space reflections exhibit fewer "breakups", at the cost of some objects creating physically incorrect reflections.

Keep in mind that screen-space-reflections only work for reflecting opaque geometry. Transparent materials won't be reflected, as they don't write to the depth buffer. This also applies to shaders that use `hint_screen_texture` or `hint_depth_texture` uniforms.

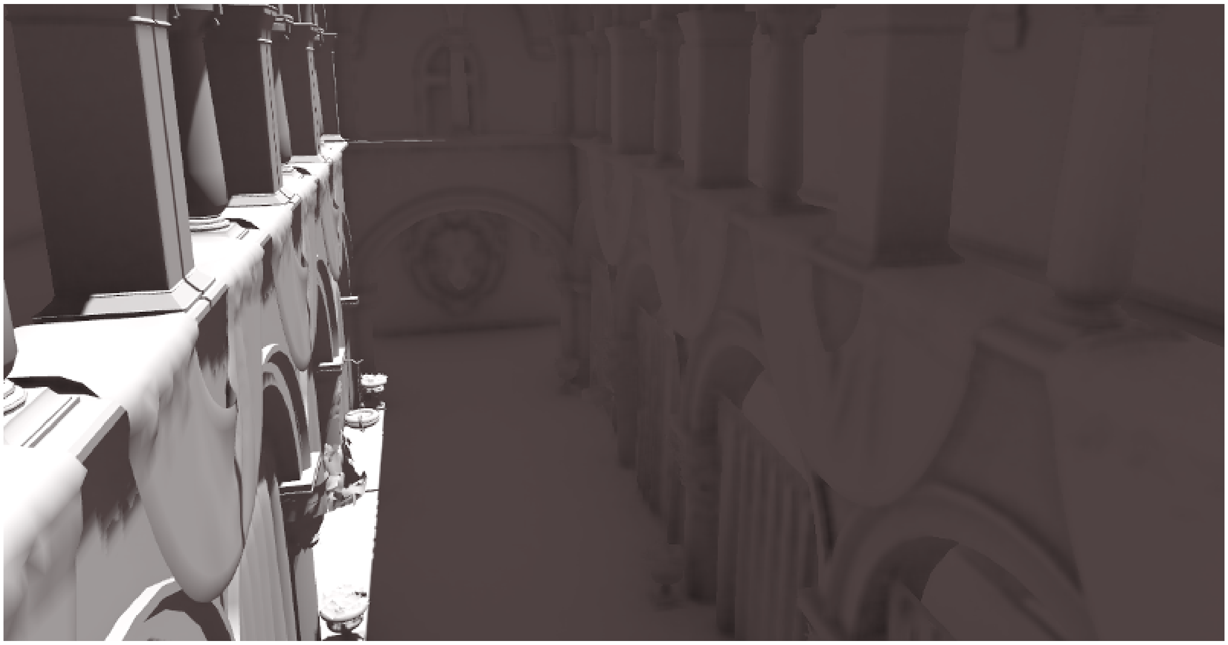
Screen-Space Ambient Occlusion (SSAO)

This feature is only available when using the Forward+ backend, not Mobile or Compatibility.

As mentioned in the **Ambient** section, areas where light from light nodes does not reach (either because it's outside the radius or shadowed) are lit with ambient light. Godot can simulate this using VoxelGI, ReflectionProbe, the Sky, or a constant ambient color. The problem, however, is that all the methods proposed previously act more on a larger scale (large regions) than at the smaller geometry level.

Constant ambient color and Sky are the same everywhere, while GI and Reflection probes have more local detail, but not enough to simulate situations where light is not able to fill inside hollow or concave features.

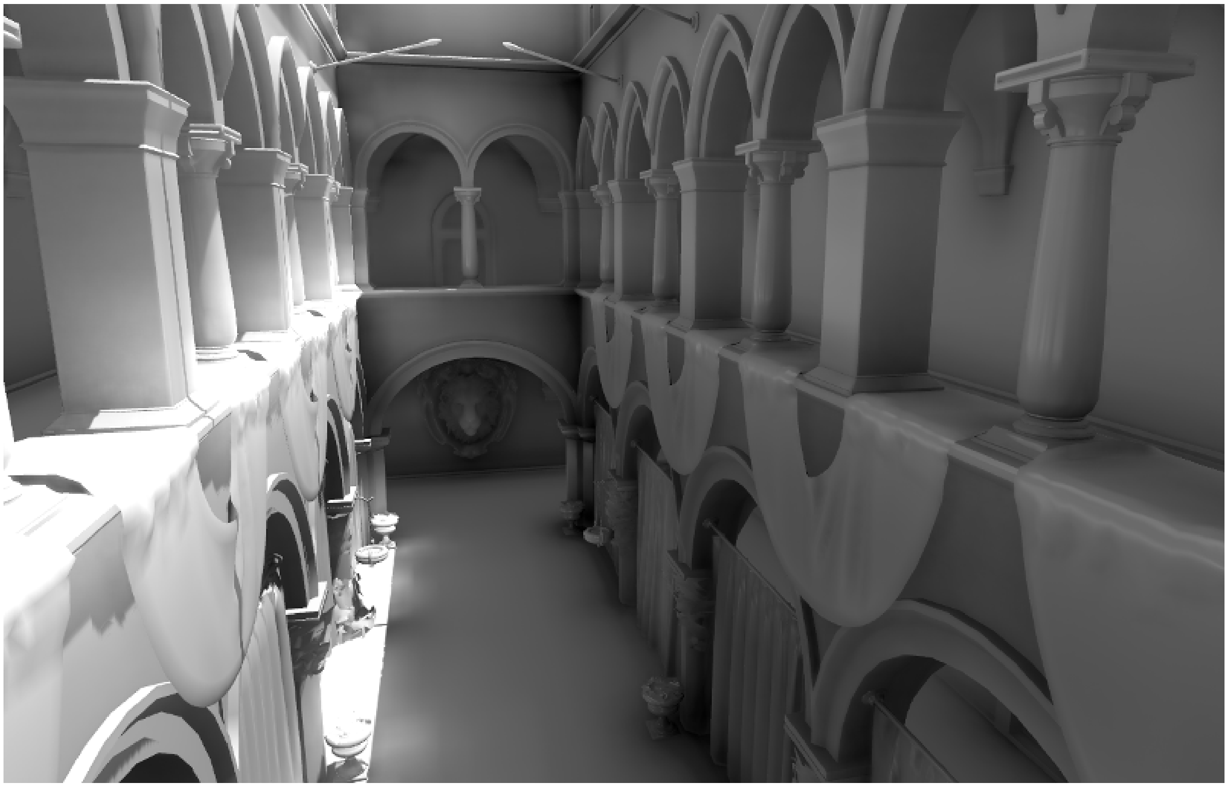
This can be simulated with Screen Space Ambient Occlusion. As you can see in the image below, its purpose is to make sure concave areas are darker, simulating a narrower path for the light to enter:



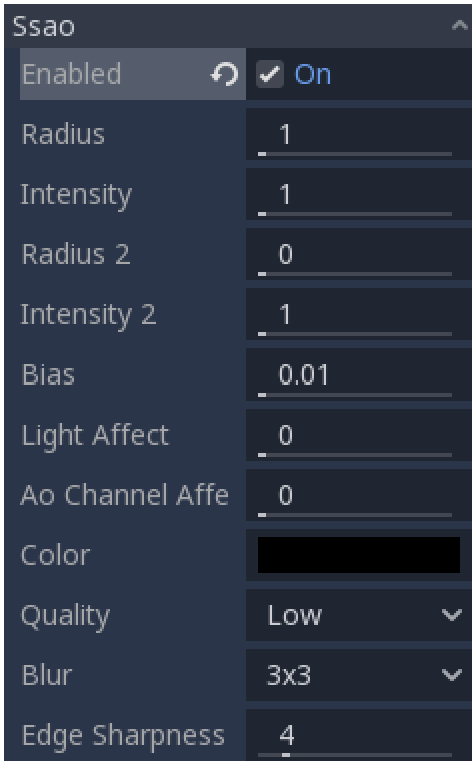
It is a common mistake to enable this effect, turn on a light, and not be able to appreciate it. This is because SSAO only acts on *ambient* light. It does not affect direct light.

This is why, in the image above, the effect is less noticeable under the direct light (on the left). If you want to force SSAO to work with direct light too, use the **Light Affect** parameter. Even though this is not physically correct, some artists like how it looks.

SSAO looks best when combined with a real source of indirect light, like VoxelGI:



Tweaking SSAO is possible with several parameters:



Ssao

Enabled	☑ On
Radius	1
Intensity	1
Radius 2	0
Intensity 2	1
Bias	0.01
Light Affect	0
Ao Channel Affe	0
Color	■
Quality	Low
Blur	3x3
Edge Sharpness	4

- **Radius:** The distance at which objects can occlude each other when calculating screen-space ambient occlusion. Higher values will result in occlusion over a greater distance at the cost of performance and quality.
- **Intensity:** The primary screen-space ambient occlusion intensity. Acts as a multiplier for the screen-space ambient occlusion effect. A higher value results in darker occlusion. Since SSAO is a screen-space effect, it's recommended to remain conservative with this value. SSAO that is too strong can be distracting during gameplay.
- **Power:** The distribution of occlusion. A higher value results in darker occlusion, similar to **Intensity**, but with a sharper falloff.
- **Detail:** Sets the strength of the additional level of detail for the screen-space ambient occlusion effect. A high value makes the detail pass more prominent, but it may contribute to aliasing in your final image.
- **Horizon:** The threshold for considering whether a given point on a surface is occluded or not represented as an angle from the horizon mapped into the 0.0-1.0 range. A value of 1.0 results in no occlusion.
- **Sharpness:** The amount that the screen-space ambient occlusion effect is allowed to blur over the edges of objects. Setting too high will result in aliasing around the edges of objects. Setting too low will make object edges appear blurry.
- **Light Affect:** The screen-space ambient occlusion intensity in direct light. In real life, ambient occlusion only applies to indirect light, which means its effects can't be seen in direct light. Values higher than 0 will make the SSAO effect visible in direct light. Values above `0.0` are not physically accurate, but some artists prefer this effect.

Screen-Space Indirect Lighting (SSIL)

This feature is only available when using the Forward+ backend, not Mobile or Compatibility.

SSIL provides indirect lighting for small details or dynamic geometry that other global illumination techniques cannot cover. This applies to bounced diffuse lighting, but also emissive materials. When SSIL is enabled on its own, the effect may not be that noticeable, which is intended.

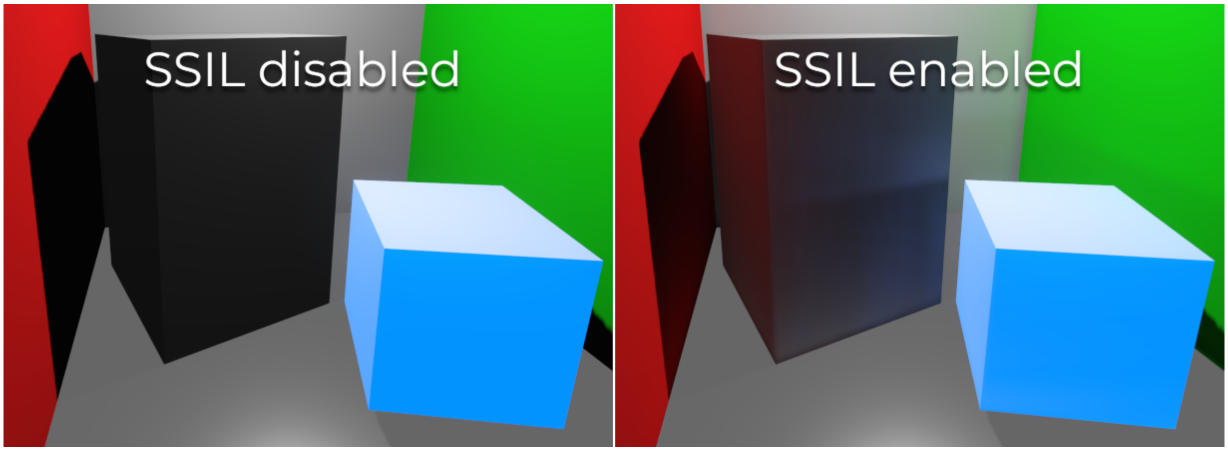
Instead, SSIL is meant to be used as a *complement* to other global illumination techniques such as VoxelGI, SDFGI and LightmapGI. SSIL also provides a subtle ambient occlusion effect, similar to SSAO but with less detail.

This feature only provides indirect lighting. It is not a full global illumination solution. This makes it different from screen-space global illumination (SSGI) offered by other 3D engines. SSIL can be combined with SSR and/or SSAO for greater visual quality (at the cost of performance).

Tweaking SSIL is possible with several parameters:

- **Radius:** The distance that bounced lighting can travel when using the screen space indirect lighting effect. A larger value will result in light bouncing further in a scene, but may result in under-sampling artifacts which look like long spikes surrounding light sources.
- **Intensity:** The brightness multiplier for the screen-space indirect lighting effect. A higher value will result in brighter light.
- **Sharpness:** The amount that the screen-space indirect lighting effect is allowed to blur over the edges of objects. Setting too high will result in aliasing around the edges of objects. Setting too low will make object edges appear blurry.
- **Normal Rejection:** Amount of normal rejection used when calculating screen-space indirect lighting. Normal rejection uses the normal of a given sample point to reject samples that are facing away from the current pixel. Normal rejection is necessary to avoid light leaking when only one side of an object is illuminated. However, normal rejection can be disabled if light

leaking is desirable, such as when the scene mostly contains emissive objects that emit light from faces that cannot be seen from the camera.



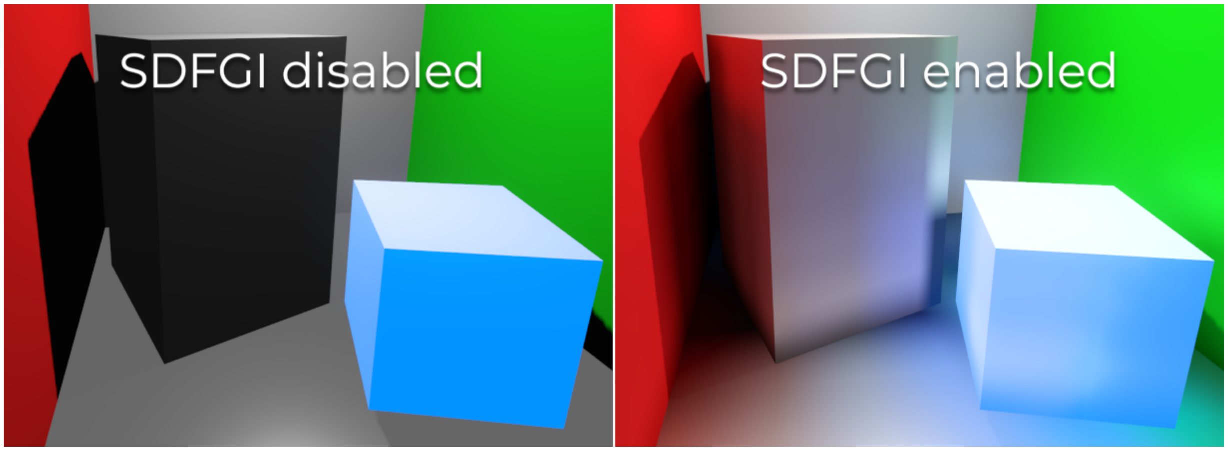
Signed Distance Field Global Illumination (SDFGI)

This feature is only available when using the Forward+ backend, not Mobile or Compatibility.

Signed distance field global illumination (SDFGI) is a form of real-time global illumination. It is not a screen-space effect, which means it can provide global illumination for off-screen elements (unlike SSIL).

See also

See Signed distance field global illumination (SDFGI) for instructions on setting up this global illumination technique.



Glow

Note

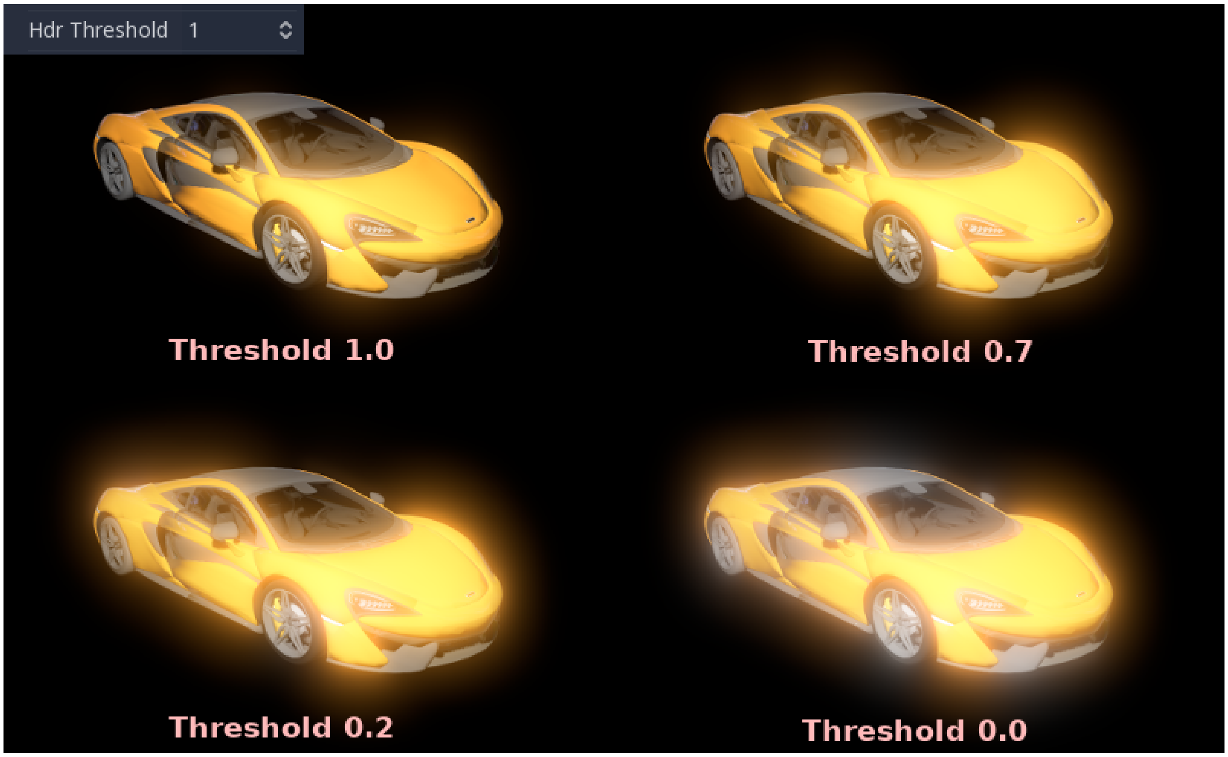
When using the Compatibility rendering method, glow uses a different implementation with some properties being unavailable and hidden from the inspector: **Levels**, **Normalized**, **Strength**, **Blend Mode**, **Mix**, **Map**, and **Map Strength**.

This implementation is optimized to run on low-end devices and is less flexible as a result.

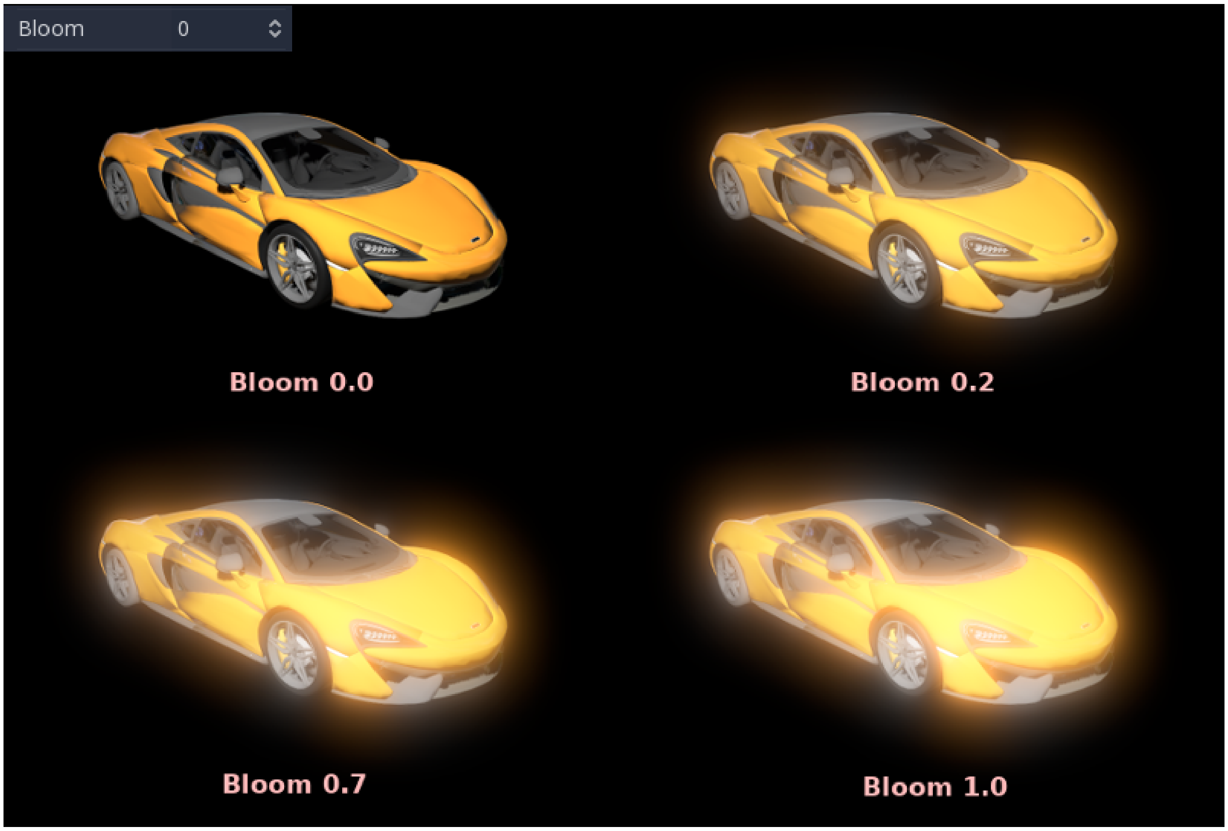
In photography and film, when light amount exceeds the maximum *luminance* (brightness) supported by the media, it generally bleeds outwards to darker regions of the image. This is simulated in Godot with the **Glow** effect.

By default, even if the effect is enabled, it will be weak or invisible. One of two conditions need to happen for it to actually show:

- 1. The light in a pixel surpasses the **HDR Threshold** (where 0 is all light surpasses it, and 1.0 is light over the tonemapper **White** value). Normally, this value is expected to be at 1.0, but it can be lowered to allow more light to bleed. There is also an extra parameter, **HDR Scale**, that allows scaling (making brighter or darker) the light surpassing the threshold.


Hdr Threshold 1
Threshold 1.0
Threshold 0.7
Threshold 0.2
Threshold 0.0

- 2. The **Bloom** property has a value greater than `0.0`. As it increases, it sends the whole screen to the glow processor at higher amounts.



Both will cause the light to start bleeding out of the brighter areas.

Once glow is visible, it can be controlled with a few extra parameters:

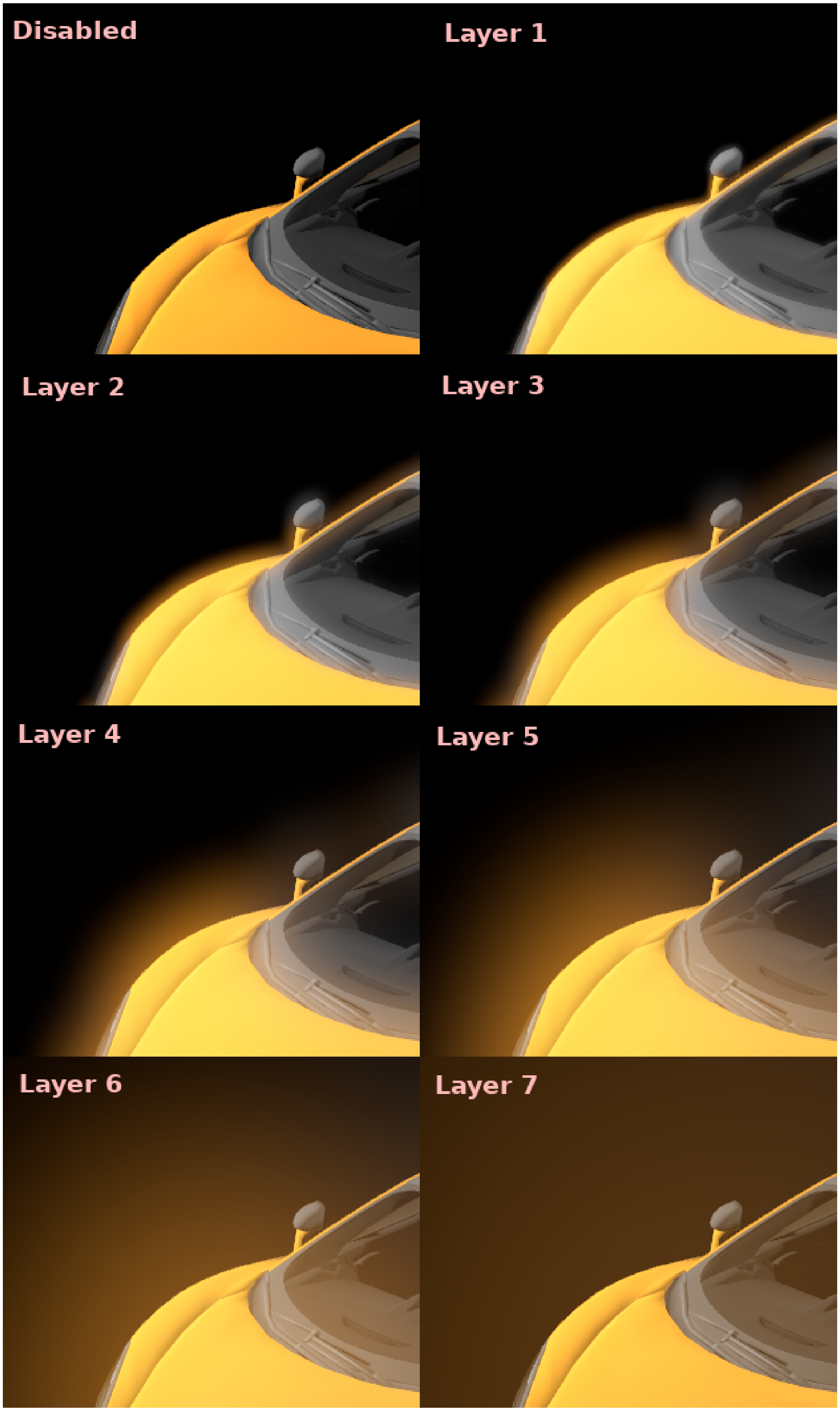
- **Intensity** is an overall scale for the effect, it can be made stronger or weaker (`0.0` removes it).
- **Strength** is how strong the gaussian filter kernel is processed. Greater values make the filter saturate and expand outwards. In general, changing this is not needed, as the size can be adjusted more efficiently with the **Levels**.

The **Blend Mode** of the effect can also be changed:

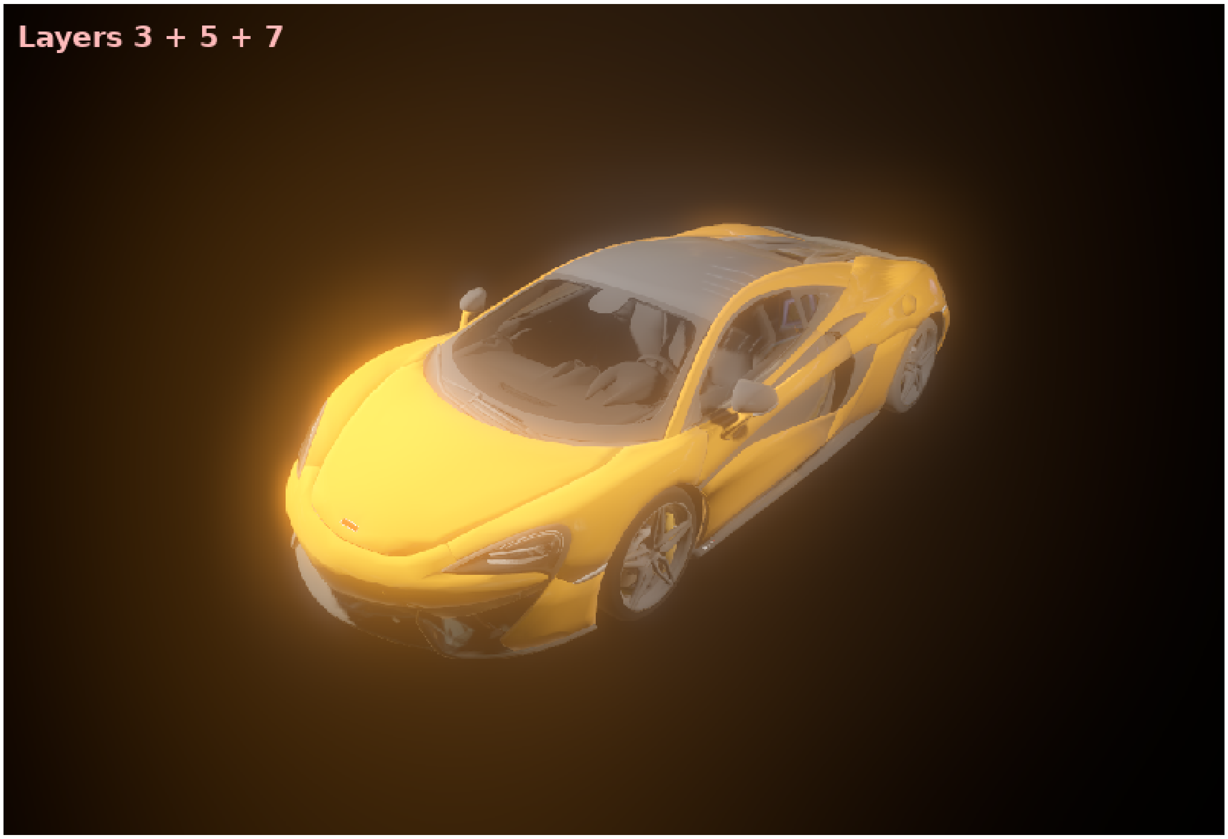
- **Additive** is the strongest one, as it only adds the glow effect over the image with no blending involved. In general, it's too strong to be used, but can look good with low-intensity **Bloom** (produces a dream-like effect).

- **Screen** ensures glow never brightens more than itself and it works great as an all around.
- **Softlight** is the default and weakest one, producing only a subtle color disturbance around the objects. This mode works best on dark scenes.
- **Replace** can be used to blur the whole screen or debug the effect. It only shows the glow effect without the image below.
- **Mix** mixes the glow effect with the main image. This can be used for greater artistic control. The mix factor is controlled by the **Mix** property which appears above the blend mode (only when the blend mode is set to Mix). High mix factor values will appear to darken the image unless **Bloom** is increased.

To change the glow effect size and shape, Godot provides **Levels**. Smaller levels are strong glows that appear around objects, while large levels are hazy glows covering the whole screen:


Disabled
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Layer 6
Layer 7

The real strength of this system, though, is to combine levels to create more interesting glow patterns:



Finally, the glow effect can be controlled using a *glow map*, which is a texture that determines how bright glow should be on each part of the screen. This texture can optionally be colored to tint the glow effect to the glow map's color. The texture is stretched to fit the viewport, so using an aspect ratio that matches your viewport's most common aspect ratio (such as 16:9) is recommended to avoid visible distortion.

There are 2 main use cases for a glow map texture:

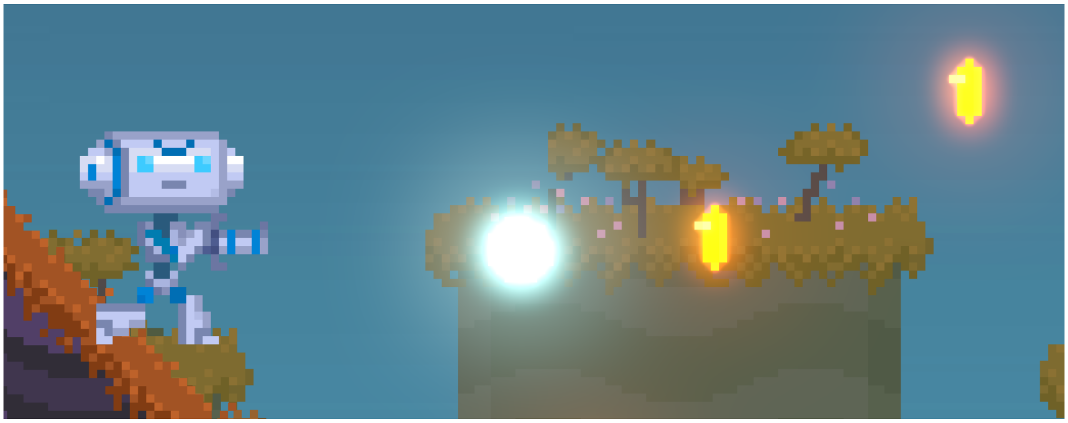
- Create a "lens dirt" effect using a dirt pattern texture.
- Make glow less strong on specific parts of the screen by using a gradient texture.

Without glow map
With glow map

Using glow in 2D

There are 2 ways to use glow in 2D:

- Since Godot 4.2, you can enable HDR for 2D rendering when using the Forward+ and Mobile rendering methods. This has a performance cost, but it allows for a greater dynamic range. This also allows you to control which objects glow using their individual **Modulate** or **Self Modulate** properties (use the RAW mode in the color picker). Enabling HDR can also reduce banding in the 2D rendering output.
  - To enable HDR in 2D, open the Project Settings, enable **Rendering > Viewport > HDR 2D** then restart the editor.
- If you want to maximize performance, you can leave HDR disabled for 2D rendering. However, you will have less control on which objects glow.
  - Enable glow, set the environment background mode to **Canvas** then decrease **Glow HDR Threshold** so that pixels that are not overbright will still glow. To prevent UI elements from glowing, make them children of a CanvasLayer node. You can control which layers are affected by glow using the **Background > Canvas Max Layer** property of the Environment resource.



Example of using glow in a 2D scene. HDR 2D is enabled, while coins and the bullet have their **Modulate** property increased to overbright values using the RAW mode in the color picker.

Warning

The 2D renderer renders in linear color space if the **Rendering > Viewport > HDR 2D** project setting is enabled, so `source_color` must also be used for uniform samplers that are used as color input in `canvas_item` shaders. If this is not done, the texture will appear washed out.

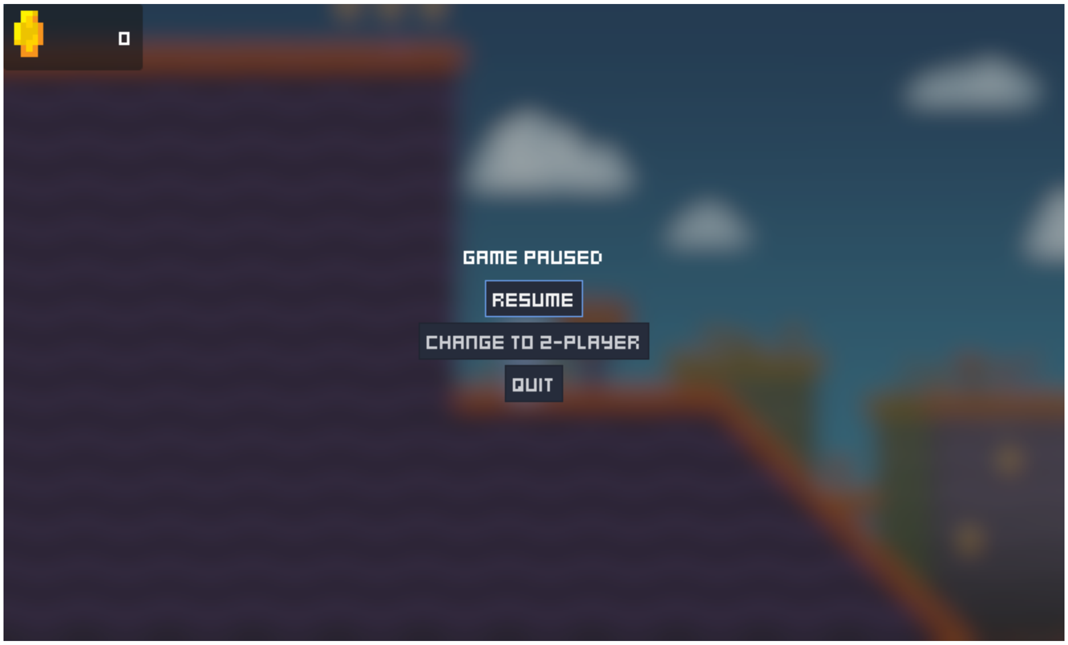
If 2D HDR is disabled, `source_color` will keep working correctly in `canvas_item` shaders, so it's recommend to use it when relevant either way.

Using glow to blur the screen

Glow can be used to blur the whole viewport, which is useful for background blur when a menu is open. Only 3D rendering will be affected unless the environment's background mode is set to **Canvas**. To prevent UI elements from being blurred when using the Canvas background mode, make them children of a CanvasLayer node. You can control which layers are affected by this blurring effect using the **Background > Canvas Max Layer** property of the Environment resource.

To use glow as a blurring solution:

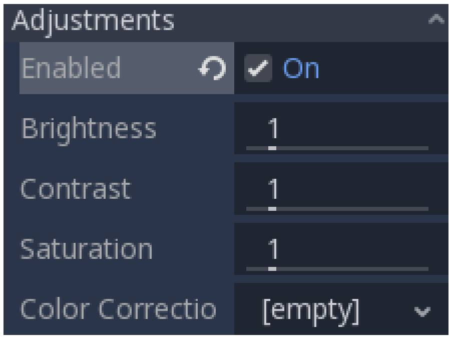
- Enable **Normalized** and adjust levels according to preference. Increasing higher level indices will result in a more blurred image. It's recommended to leave a single glow level at `1.0` and leave all other glow levels at `0.0`, but this is not required. Note that the final appearance will vary depending on viewport resolution.
- Set **Intensity** to `1.0` and **Bloom** to `1.0`.
- Set the blend mode to **Replace** and **HDR Luminance Cap** to `1.0`.



Example of using glow to blur the 2D rendering in the menu's background

Adjustments

At the end of processing, Godot offers the possibility to do some standard image adjustments.



Adjustments

Enabled ☑ On

Brightness 1

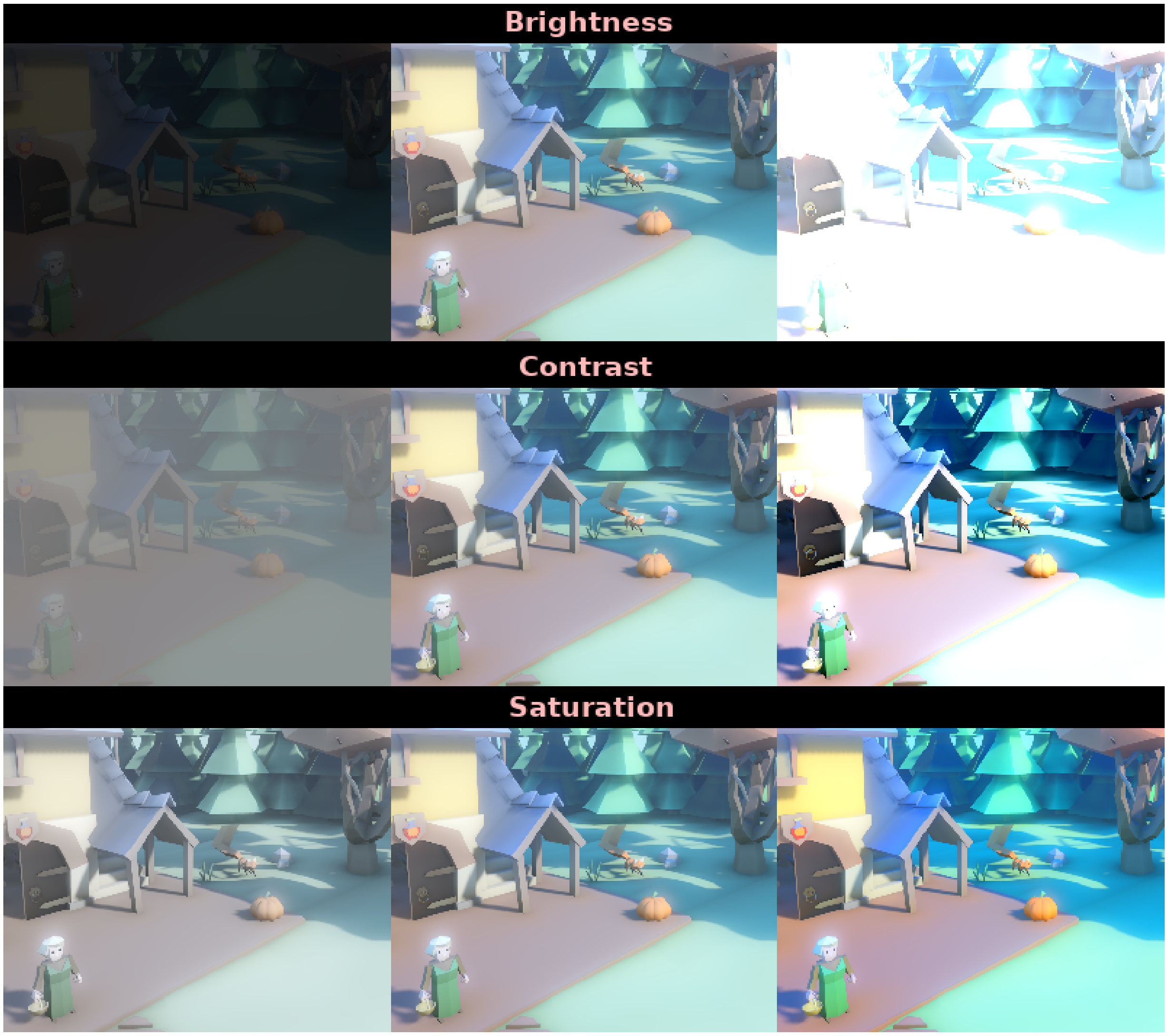
Contrast 1

Saturation 1

Color Correctio [empty]

Basic BCS adjustments

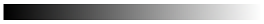
The first adjustment is being able to change the typical **Brightness**, **Contrast**, and **Saturation** properties:



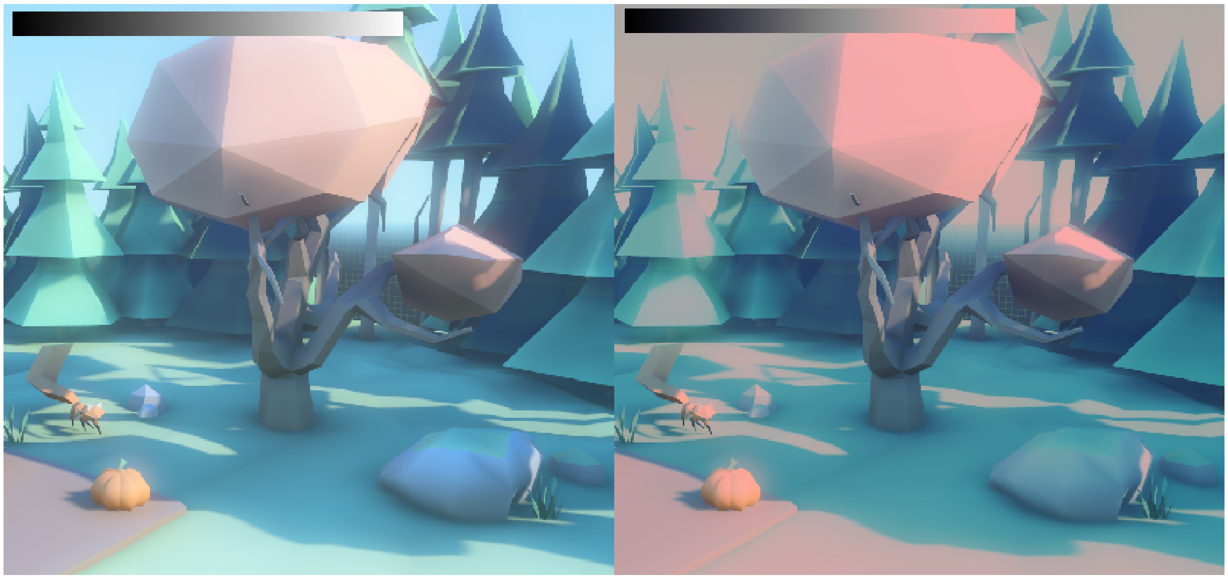
Color correction using a 1D gradient

The second adjustment is by supplying a color correction gradient. This can be done by assigning a GradientTexture1D resource to the **Color Correction** property, or by loading a texture containing a horizontal gradient. The leftmost part of the gradient represents black in the source image, whereas the rightmost part of the gradient represents white in the source image.

A linear black-to-white gradient like the following one will produce no effect:



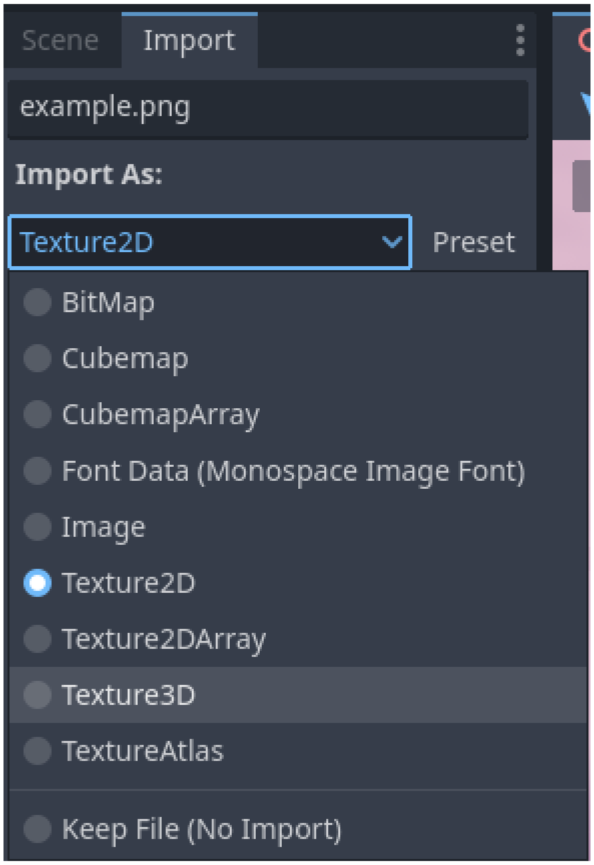
But creating custom ones will allow to map each channel to a different color:



Color correction using a 3D LUT

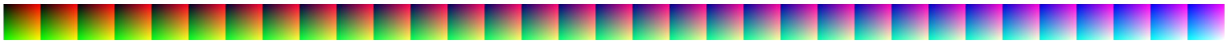
A 3D look-up-texture (LUT) can also be used for color correction. This is a special texture used to modify each color channel separately from one another (red, green, blue). This image can be of any resolution, but since color correction is low-frequency data, sticking to low resolutions is recommended for performance reasons. A LUT texture's resolution is typically 17×17×17, 33×33×33, 51×51×51 or 65×65×65 (the odd size allows for better interpolation).

For this to work, the look-up texture's import mode must be set to Texture3D in the Import dock (instead of being imported as a regular Texture2D):


Scene
Import
example.png
Import As:
Texture2D
Preset
BitMap
Cubemap
CubemapArray
Font Data (Monospace Image Font)
Image
Texture2D
Texture2DArray
Texture3D
TextureAtlas
Keep File (No Import)

Make sure to configure the number of horizontal and vertical slices to import as well. If you don't do this, the LUT texture will not affect the viewport correctly when used. You can preview how the 3D texture was imported by double-clicking it, in the FileSystem dock, then going to the inspector to flip through the texture's layers.

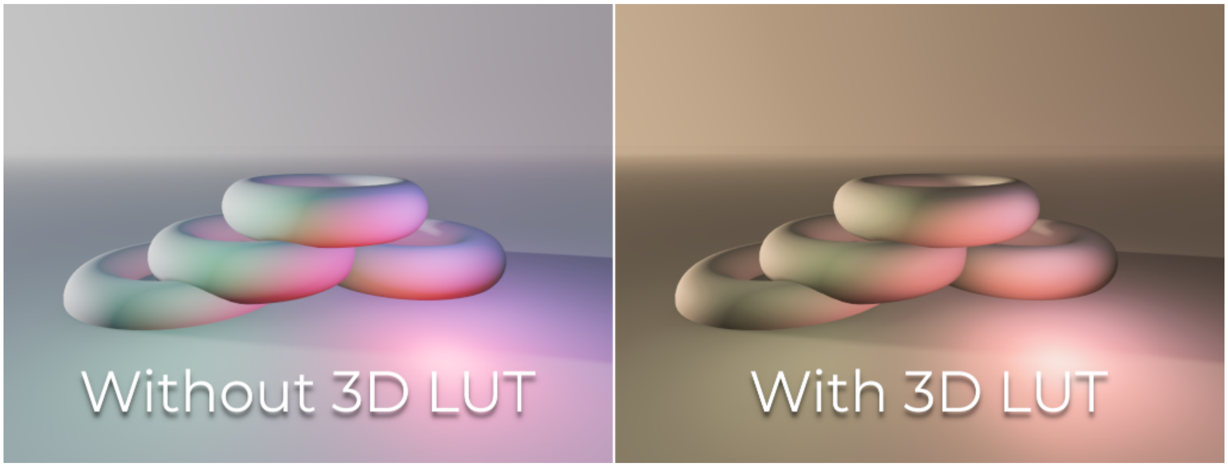
You can use this neutral 33×33×33 LUT template as a base (right-click and choose **Save as...**):



With the above LUT template, after changing its import mode to **Texture3D**, set its number of **Horizontal** slices to `33` in the Import dock then click **Reimport**. If you load this LUT into the **Color Correction** property, you won't see any visible difference for now since this texture is designed to be a neutral starting point.

This LUT template can be modified in an image editor to provide a different mood to the image. A common workflow is to place the LUT image next to a screenshot of the project's 3D viewport, then use an image editor to modify both the LUT image and the screenshot at the same time. The LUT can then be saved and applied to the game engine to perform the same color correction in real-time.

For example, modifying the LUT template in an image editor to give it a "sepia" look results in the image on the right:



Note

Adjustments and color correction are applied *after* tonemapping. This means the tonemapping properties defined above still have an effect when adjustments are enabled.

Camera attribute options

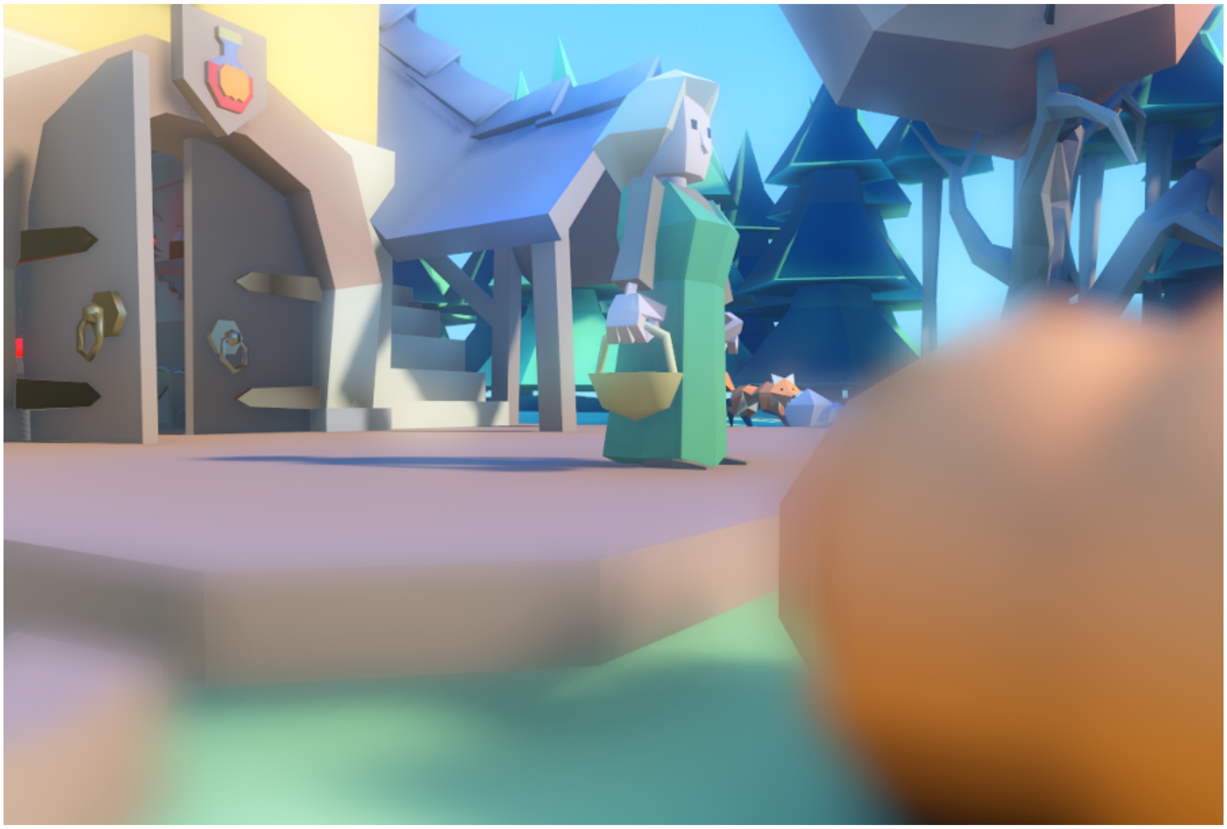
Depth of Field / Far Blur

This effect simulates focal distance on cameras. It blurs objects behind a given range. It has an initial **Distance** with a **Transition** region (in world units):

The **Amount** parameter controls the amount of blur. For larger blurs, tweaking the depth of field quality in the advanced project settings may be needed to avoid artifacts.

Depth of Field / Near Blur

This effect simulates focal distance on cameras. It blurs objects close to the camera (acts in the opposite direction as far blur). It has an initial **Distance** with a **Transition** region (in world units):



The **Amount** parameter controls the amount of blur. For larger blurs, tweaking the **Quality** may be needed in order to avoid artifacts.

It is common to use both blurs together to focus the viewer's attention on a given object, or create a so-called "tilt shift" effect [https://en.wikipedia.org/wiki/Miniature_faking].

Note

When using CameraAttributesPhysical instead of CameraAttributesPractical, depth of field is automatically computed from the camera attributes' focus distance, focal length, and aperture.

Exposure

This multiplies the overall scene brightness visible from the camera. Higher values result in a visually brighter scene.

Auto Exposure

This feature is only available when using the Forward+ backend, not Mobile or Compatibility.

Even though, in most cases, lighting and texturing are heavily artist controlled, Godot supports a basic high dynamic range implementation with the auto exposure mechanism. This is generally used to add realism when combining interior areas with low light and bright outdoor areas. Auto exposure simulates the camera (or eye) in an effort to adapt between light and dark locations and their different amounts of light.

Note

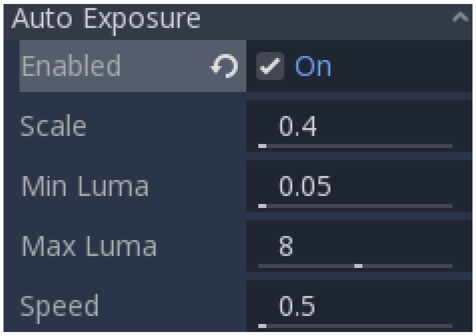
Auto exposure needs to evaluate the scene's brightness every frame, which has a moderate performance cost. Therefore, it's recommended to leave Auto Exposure disabled if it doesn't make much of a difference in your scene.

tutorials/3d/img/environment_hdr_autoexp.png

The simplest way to use auto exposure is to make sure outdoor lights (or other strong lights) have energy beyond 1.0. This is done by tweaking their **Energy** multiplier (on the Light itself). To make it consistent, the **Sky** usually needs to use the energy multiplier too, to match with the directional light. Normally, values between 3.0 and 6.0 are enough to simulate indoor-outdoor conditions.

By combining Auto Exposure with Glow post-processing, pixels that go over the tonemap **White** will bleed to the glow buffer, creating the typical bloom effect in photography.

The user-controllable values in the Auto Exposure section come with sensible defaults, but you can still tweak them:



Auto Exposure

Enabled: ☑ On

Scale: 0.4

Min Luma: 0.05

Max Luma: 8

Speed: 0.5

- **Scale:** Value to scale the lighting. Higher values produce brighter images, and lower values produce darker ones.
- **Min Sensitivity / Min Exposure Value:** Minimum luminance that auto exposure will aim to adjust for (in ISO when using CameraAttributesPractical, or in EV100 when using CameraAttributesPhysical). Luminance is the average of the light in all the pixels of the screen.

- **Max Sensitivity / Max Exposure Value:** Maximum luminance that auto exposure will aim to adjust for (in ISO when using CameraAttributesPractical, or in EV100 when using CameraAttributesPhysical).
- **Speed:** Speed at which luminance corrects itself. The higher the value, the faster luminance correction happens. High values may be more suited to fast-paced games, but can be distracting in some scenarios.

When using CameraAttributesPractical, exposure is set using *sensitivity* defined in ISO instead of an exposure value in EV100. Typical ISO values are between 50 and 3200, with higher values resulting in higher final exposure. In real life, daytime photography generally uses ISO values between 100 and 800.

See also

See Physical light and camera units if you wish to use real world units to configure your camera's exposure, field of view and depth of field.

Volumetric fog and fog volumes

Note

Volumetric fog is only supported in the Clustered Forward rendering backend, not Forward Mobile or Compatibility.

As described in Environment and post-processing, Godot supports various visual effects including two types of fog: traditional (non-volumetric) fog and volumetric fog. Traditional fog affects the entire scene at once and cannot be customized with Fog shaders.

Volumetric fog can be used at the same time as non-volumetric fog if desired.

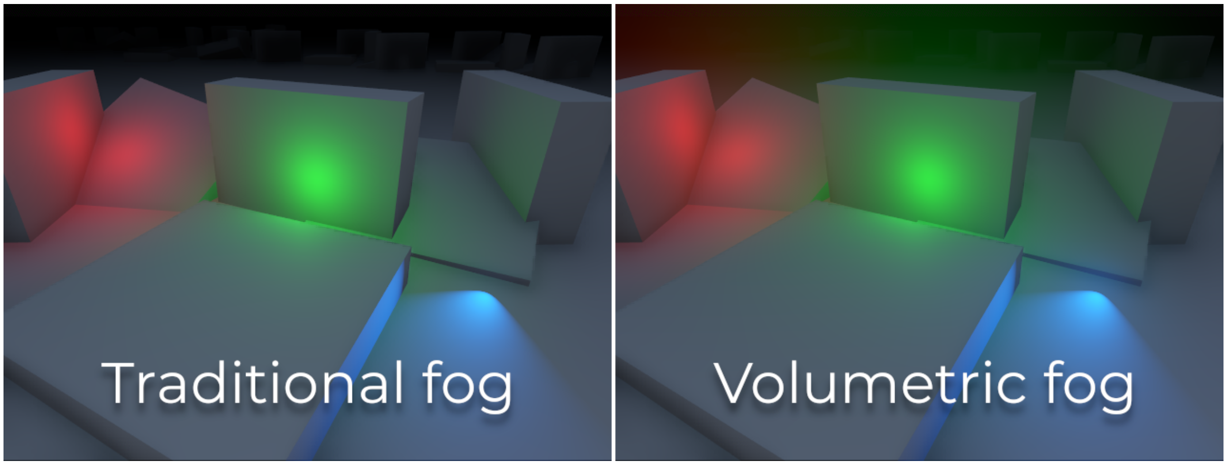
On this page, you'll learn:

- How to set up volumetric fog in Godot.
- What fog volumes are and how they differ from "global" volumetric fog.

See also

You can see how volumetric fog works in action using the Volumetric Fog demo project [https://github.com/godotengine/godot-demo-projects/tree/master/3d/volumetric_fog].

Here is a comparison between traditional fog (which does not interact with lighting) and volumetric fog, which is able to interact with lighting:



Volumetric fog properties

After enabling volumetric fog in the WorldEnvironment node's Environment resource, you can edit the following properties:

- **Density:** The base *exponential* density of the volumetric fog. Set this to the lowest density you want to have globally. FogVolumes can be used to add to or subtract from this density in specific areas. A value of `0.0` disables global volumetric fog while allowing FogVolumes to display volumetric fog in specific areas. Fog rendering is exponential as in real life.
- **Albedo:** The Color of the volumetric fog when interacting with lights. Mist and fog have an albedo close to white (`Color(1, 1, 1, 1)`) while smoke has a darker albedo.
- **Emission:** The emitted light from the volumetric fog. Even with emission, volumetric fog will not cast light onto other surfaces. Emission is useful to establish an ambient color. As the volumetric fog effect uses single-scattering only, fog tends to need a little bit of emission to soften the harsh shadows.
- **Emission Energy:** The brightness of the emitted light from the volumetric fog.
- **GI Inject:** Scales the strength of Global Illumination used in the volumetric fog's albedo color. A value of `0.0` means that

Global Illumination will not impact the volumetric fog. This has a small performance cost when set above `0.0`.

- **Anisotropy:** The direction of scattered light as it goes through the volumetric fog. A value close to `1.0` means almost all light is scattered forward. A value close to `0.0` means light is scattered equally in all directions. A value close to `-1.0` means light is scattered mostly backward. Fog and mist scatter light slightly forward, while smoke scatters light equally in all directions.
- **Length:** The distance over which the volumetric fog is computed. Increase to compute fog over a greater range, decrease to add more detail when a long range is not needed. For best quality fog, keep this as low as possible.
- **Detail Spread:** The distribution of size down the length of the froxel buffer. A higher value compresses the froxels closer to the camera and places more detail closer to the camera.
- **Ambient Inject:** Scales the strength of ambient light used in the volumetric fog. A value of `0.0` means that ambient light will not impact the volumetric fog. This has a small performance cost when set above `0.0`.
- **Sky Affect:** Controls how much volumetric fog should be drawn onto the background sky. If set to `0.0`, volumetric fog won't affect sky rendering at all (including FogVolumes).

Two additional properties are offered in the **Temporal Reprojection** section:

- **Temporal Reprojection > Enabled:** Enables temporal reprojection in the volumetric fog. Temporal reprojection blends the current frame's volumetric fog with the last frame's volumetric fog to smooth out jagged edges. The performance cost is minimal, however it does lead to moving FogVolumes and Light3Ds "ghosting" and leaving a trail behind them. When temporal reprojection is enabled, try to avoid moving FogVolumes or Light3Ds too fast. Short-lived dynamic lighting effects should have **Volumetric Fog Energy** set to `0.0` to avoid ghosting.

- **Temporal Reprojection > Amount:** The amount by which to blend the last frame with the current frame. A higher number results in smoother volumetric fog, but makes "ghosting" much worse. A lower value reduces ghosting but can result in the per-frame temporal jitter becoming visible.

Note

Unlike non-volumetric fog, volumetric fog has a *finite* range. This means volumetric fog cannot entirely cover a large world, as it will eventually stop being rendered in the distance.

If you wish to hide distant areas from the player, it's recommended to enable both non-volumetric fog and volumetric fog at the same time, and adjust their density accordingly.

Light interaction with volumetric fog

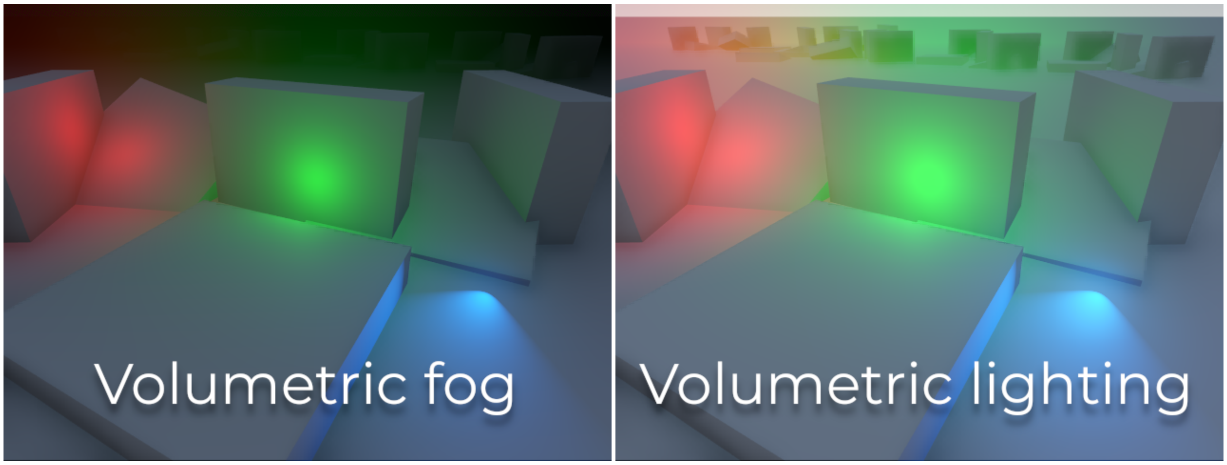
To simulate fog light scattering behavior in real life, all light types will interact with volumetric fog. How much each light will affect volumetric fog can be adjusted using the **Volumetric Fog Energy** property on each light. Enabling shadows on a light will also make those shadows visible on volumetric fog.

If fog light interaction is not desired for artistic reasons, this can be globally disabled by setting **Volumetric Fog > Albedo** to a pure black color in the Environment resource. Fog light interaction can also be disabled for specific lights by setting its **Volumetric Fog Energy** to `0`. Doing so will also improve performance slightly by excluding the light from volumetric fog computations.

Using volumetric fog as a volumetric lighting solution

While not physically accurate, it is possible to tune volumetric fog's settings to work as volumetric *lighting* solution. This means that unlit parts of the environment will not be darkened anymore by fog, but light will still be able to make fog brighter in specific areas.

This can be done by setting volumetric fog density to the lowest permitted value *greater than zero* (`0.0001`), then increasing the **Volumetric Fog Energy** property on lights to much higher values than the default to compensate. Values between `10000` and `100000` usually work well for this.



Balancing performance and quality

There are a few project settings available to adjust volumetric fog performance and quality:

- **Rendering > Environment > Volumetric Fog > Volume Size:** Base size used to determine size of froxel buffer in the camera X-axis and Y-axis. The final size is scaled by the aspect ratio of the screen, so actual values may differ from what is set.

Set a larger size for more detailed fog, set a smaller size for better performance.

- **Rendering > Environment > Volumetric Fog > Volume Depth:** Number of slices to use along the depth of the froxel buffer for volumetric fog. A lower number will be more efficient but may result in artifacts appearing during camera movement.
- **Rendering > Environment > Volumetric Fog > Use Filter:** Enables filtering of the volumetric fog effect prior to integration. This substantially blurs the fog which reduces fine details but also smooths out harsh edges and aliasing artifacts. Disable when more detail is required.

Note

Volumetric fog can cause banding to appear on the viewport, especially at higher density levels. See Color banding for guidance on reducing banding.

Using fog volumes for local volumetric fog

Sometimes, you want fog to be constrained to specific areas. Conversely, you may want to have global volumetric fog but fog should be excluded from certain areas. Both approaches can be followed using FogVolume nodes.

Here's a quick start guide to using FogVolumes:

- Make sure **Volumetric Fog** is enabled in the Environment properties. If global volumetric fog is undesired, set its **Density** to `0.0`.
- Create a FogVolume node.

- Assign a new FogMaterial to the FogVolume node's **Material** property.
- In the FogMaterial, set **Density** to a positive value to increase density within the FogVolume, or a negative value to subtract the density from global volumetric fog.
- Configure the FogVolume's extents and shape as needed.

Note

Thin fog volumes may appear to flicker when the camera moves or rotates. This can be alleviated by increasing the **Rendering > Environment > Volumetric Fog > Volume Depth** project setting (at a performance cost) or by decreasing **Length** in the Environment volumetric fog properties (at no performance cost, but at the cost of lower fog range). Alternatively, the FogVolume can be made thicker and use a lower density in the **Material**.

FogVolume properties

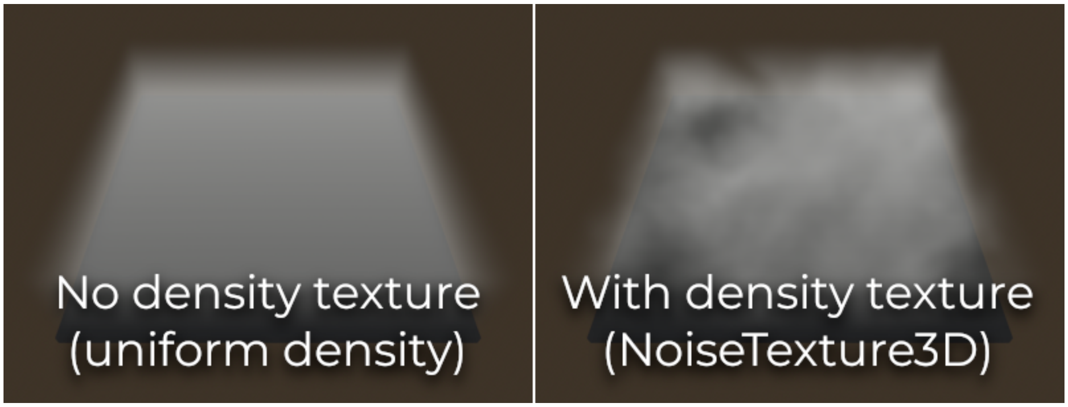
- **Extents:** The size of the FogVolume when **Shape** is **Ellipsoid**, **Cone**, **Cylinder** or **Box**. If **Shape** is **Cone** or **Cylinder**, the cone/cylinder will be adjusted to fit within the extents. Non-uniform scaling of cone/cylinder shapes via the **Extents** property is not supported, but you can scale the FogVolume node instead.
- **Shape:** The shape of the FogVolume. This can be set to **Ellipsoid**, **Cone**, **Cylinder**, **Box** or **World** (acts as global volumetric fog).
- **Material:** The material used by the FogVolume. Can be either a built-in FogMaterial or a custom ShaderMaterial (Fog shaders).

After choosing **New FogMaterial** in the **Material** property, you can adjust the following properties in FogMaterial:

- **Density:** The density of the FogVolume. Denser objects are more opaque, but may suffer from under-sampling artifacts that look like stripes. Negative values can be used to subtract fog from other FogVolumes or global volumetric fog.
- **Albedo:** The single-scattering Color of the FogVolume. Internally, member albedo is converted into single-scattering, which is additively blended with other FogVolumes and global volumetric fog's **Albedo**.
- **Emission:** The Color of the light emitted by the FogVolume. Emitted light will not cast light or shadows on other objects, but can be useful for modulating the Color of the FogVolume independently from light sources.
- **Height Falloff:** The rate by which the height-based fog decreases in density as height increases in world space. A high falloff will result in a sharp transition, while a low falloff will result in a smoother transition. A value of `0.0` results in uniform-density fog. The height threshold is determined by the height of the associated FogVolume.
- **Edge Fade:** The hardness of the edges of the FogVolume. A higher value will result in softer edges, while a lower value will result in harder edges.
- **Density Texture:** The 3D texture that is used to scale the member density of the FogVolume. This can be used to vary fog density within the FogVolume with any kind of static pattern. For animated effects, consider using a custom fog shader. You can import any image as a 3D texture by changing its import type in the Import dock.

Using 3D noise density textures

Since Godot 4.1, there is a NoiseTexture3D resource that can be used to procedurally generate 3D noise. This is well-suited to FogMaterial density textures, which can result in more detailed fog effects:


No density texture (uniform density)
With density texture (NoiseTexture3D)

Screenshot taken with **Volume Size** project setting set to 192 to make high-frequency detail more visible in the fog.

To do so, select the **Density Texture** property and choose **New NoiseTexture3D**. Edit this NoiseTexture3D by clicking it, then click **Noise** at the bottom of the NoiseTexture3D properties and choose **New FastNoiseLite**. Adjust the noise texture's width, height and depth according to your fog volume's dimensions.

To improve performance, it's recommended to use low texture sizes (64×64×64 or lower), as high-frequency detail is difficult to notice in a FogVolume. If you wish to represent more detailed density variations, you will need to increase **Rendering > Environment > Volumetric Fog > Volume Size** in the project settings, which has a performance cost.

Note

NoiseTexture3D's **Color Ramp** affects FogMaterial density textures, but since only the texture's red channel is sampled, only the color ramp's red channel will affect the resulting density.

However, using a color ramp will *not* tint the fog volume according to the texture. You would need to use a custom shader that reads a Texture3D to achieve this.

Custom FogVolume shaders

This page only covers the built-in settings offered by FogMaterial. If you need to customize fog behavior within a FogVolume node (such as creating animated fog), FogVolume nodes' appearance can be customized using Fog shaders.

Faking volumetric fog using quads

In some cases, it may be better to use specially configured QuadMeshes as an alternative to volumetric fog:

- Quads work with any rendering method, including Forward Mobile and Compatibility.
- Quads do not require temporal reprojection to look smooth, which makes them suited to fast-moving dynamic effects such as lasers. They can also represent small details which volumetric fog cannot do efficiently.
- Quads generally have a lower performance cost than volumetric fog.

This approach has a few downsides though:

- The fog effect has less realistic falloff, especially if the camera enters the fog.
- Transparency sorting issues may occur when sprites overlap.
- Performance will not necessarily be better than volumetric fog if there are lots of sprites close to the camera.

To create a QuadMesh-based fog sprite:

1. Create a MeshInstance3D node with a QuadMesh resource in the **Mesh** property. Set the size as desired.

2. Create a new StandardMaterial3D in the mesh's **Material** property.
3. In the StandardMaterial3D, set **Shading > Shading Mode** to **Unshaded**, **Billboard > Mode** to **Enabled**, enable **Proximity Fade** and set **Distance Fade** to **Pixel Alpha**.
4. Set the **Albedo > Texture** to the texture below (right-click and choose **Save as…**):
5. *After* setting the albedo texture, go to the Import dock, select the texture and change its compression mode to **Lossless** to improve quality.

The fog's color is set using the **Albedo > Color** property; its density is set using the color's alpha channel. For best results, you will have to adjust **Proximity Fade > Distance** and **Distance Fade > Max Distance** depending on the size of your QuadMesh.

Optionally, billboarding may be left disabled if you place the quad in a way where all of its corners are in solid geometry. This can be useful for fogging large planes that the camera cannot enter, such as bottomless pits.

3D antialiasing

See also

Godot also supports antialiasing in 2D rendering. This is covered on the 2D antialiasing page.

Introduction

Due to their limited resolution, scenes rendered in 3D can exhibit aliasing artifacts. These artifacts commonly manifest as a "staircase" effect on surface edges (edge aliasing) and as flickering and/or sparkles on reflective surfaces (specular aliasing).

In the example below, you can notice how edges have a blocky appearance. The vegetation is also flickering in and out, and thin lines on top of the box have almost disappeared:

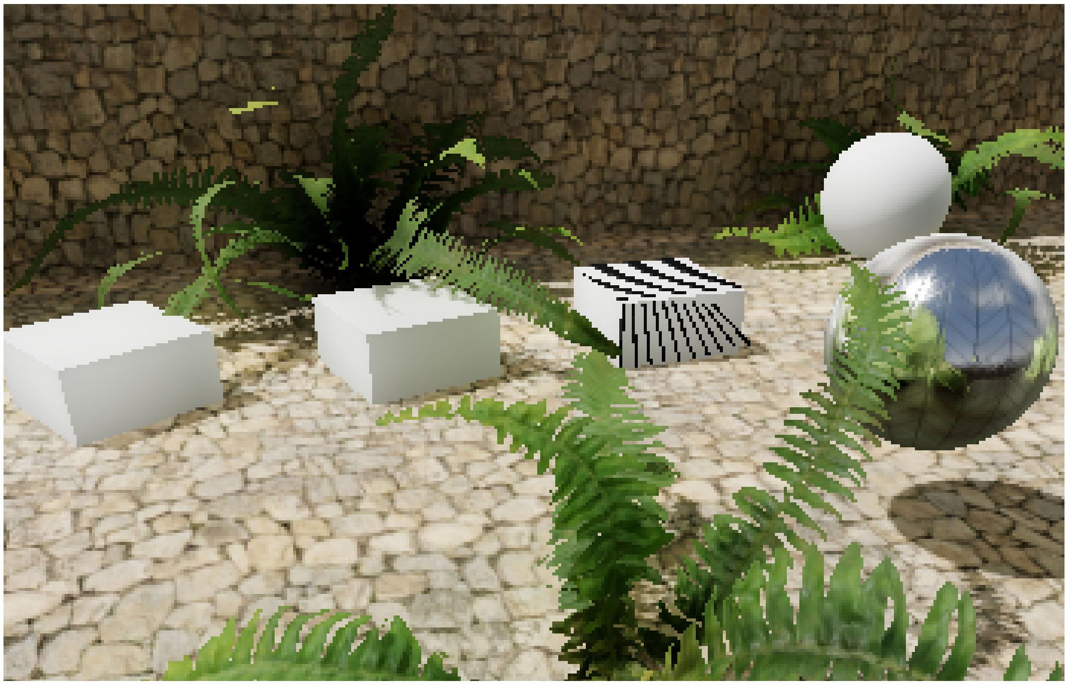


Image is scaled by 2× with nearest-neighbor filtering to make aliasing more noticeable.

To combat this, various antialiasing techniques can be used in Godot. These are detailed below.

See also

You can compare antialiasing algorithms in action using the 3D Antialiasing demo project [https://github.com/godotengine/godot-demo-projects/tree/master/3d/antialiasing].

Multisample antialiasing (MSAA)

This technique is the "historical" way of dealing with aliasing. MSAA is very effective on geometry edges (especially at higher levels). MSAA does not introduce any blurriness whatsoever.

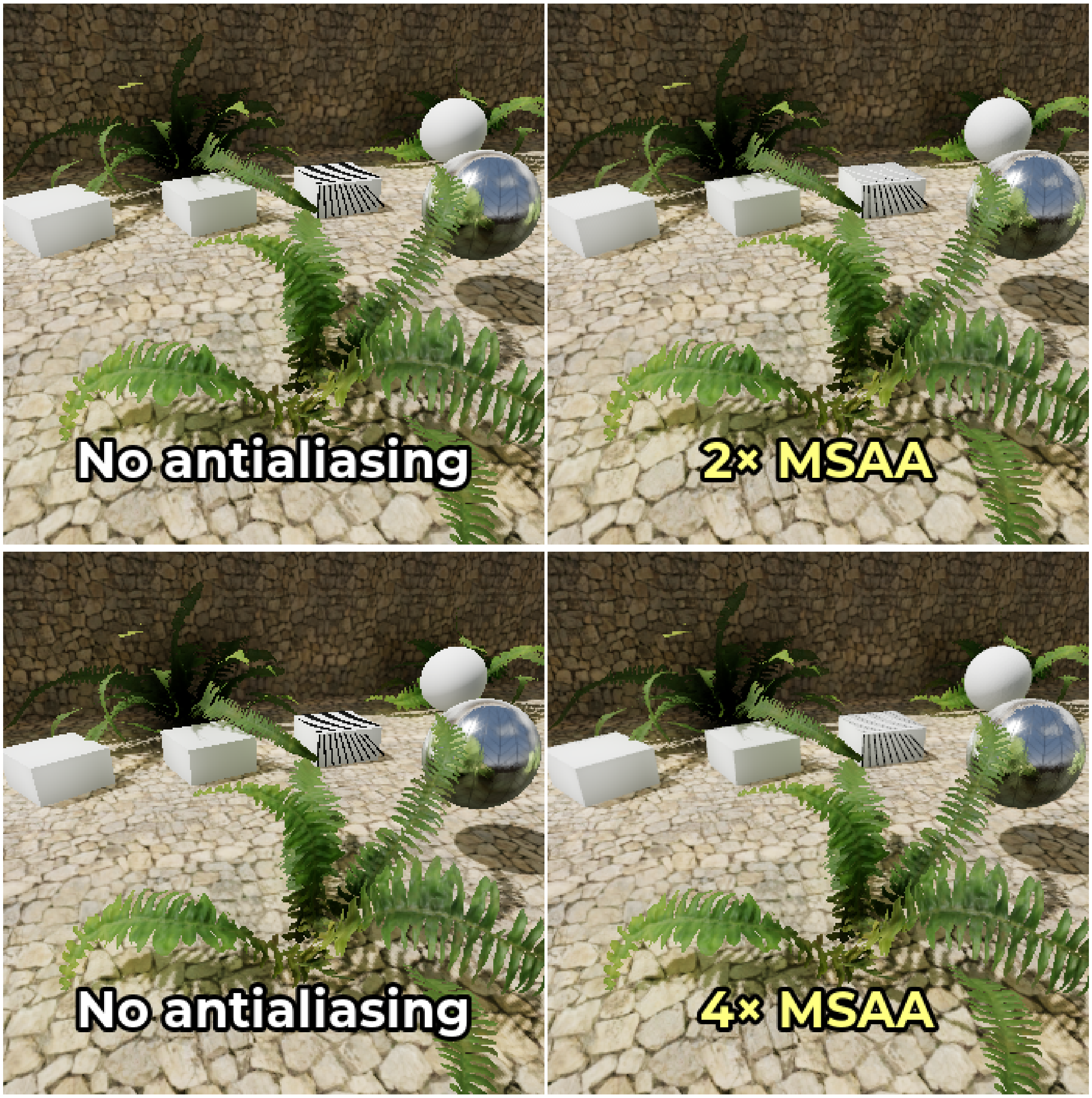
MSAA is available in 3 levels: 2×, 4×, 8×. Higher levels are more effective at antialiasing edges, but are significantly more demanding. In games with modern visuals, sticking to 2× or 4× MSAA is highly recommended as 8× MSAA is usually too demanding.

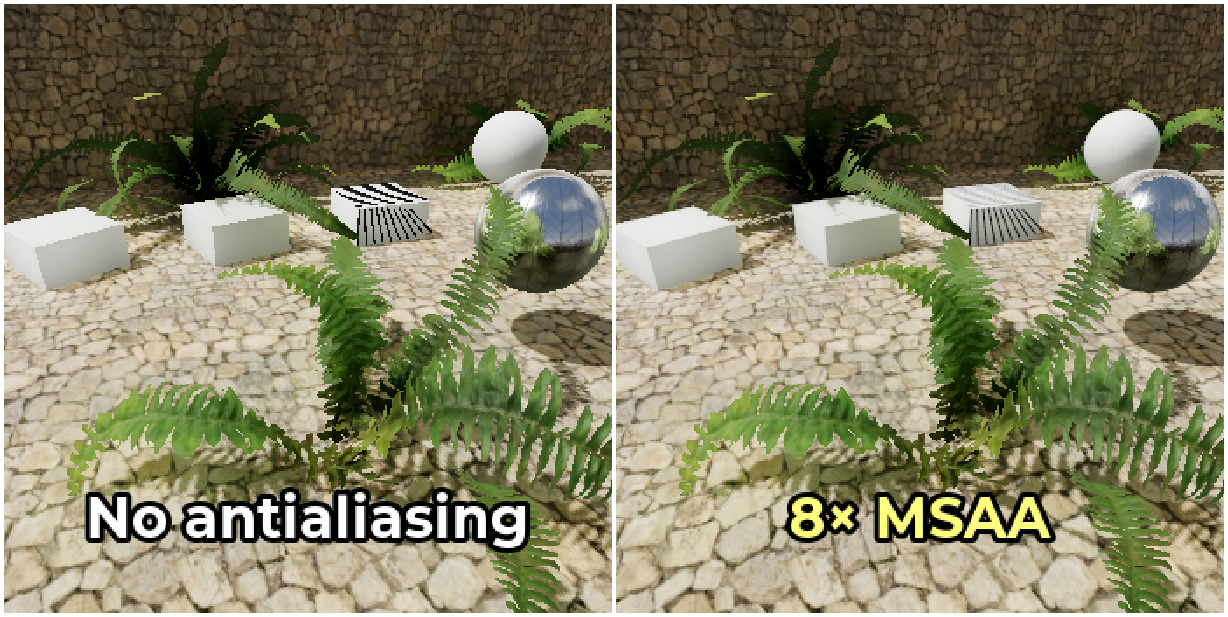
The downside of MSAA is that it only operates on edges. This is because MSAA increases the number of *coverage* samples, but not the number of *color* samples. However, since the number of color samples did not increase, fragment shaders are still run for each pixel only once. Therefore, MSAA does not reduce transparency aliasing for materials using the **Alpha Scissor** transparency mode (1-bit transparency). MSAA is also ineffective on specular aliasing.

To mitigate aliasing on alpha scissor materials, alpha antialiasing (also called *alpha to coverage*) can be enabled on specific materials in the StandardMaterial3D or ORMMaterial3D properties. Alpha to coverage has a moderate performance cost, but it's effective at reducing aliasing on transparent materials without introducing any blurriness.

MSAA can be enabled in the Project Settings by changing the value of the **Rendering > Anti Aliasing > Quality > MSAA 3D** setting. It's important to change the value of the **MSAA 3D** setting and not **MSAA 2D**, as these are entirely separate settings.

Comparison between no antialiasing (left) and various MSAA levels (right). Note that alpha antialiasing is not used here:


No antialiasing
2× MSAA
No antialiasing
4× MSAA



Temporal antialiasing (TAA)

This is only available in the Clustered Forward backend, not the Forward Mobile or Compatibility backends.

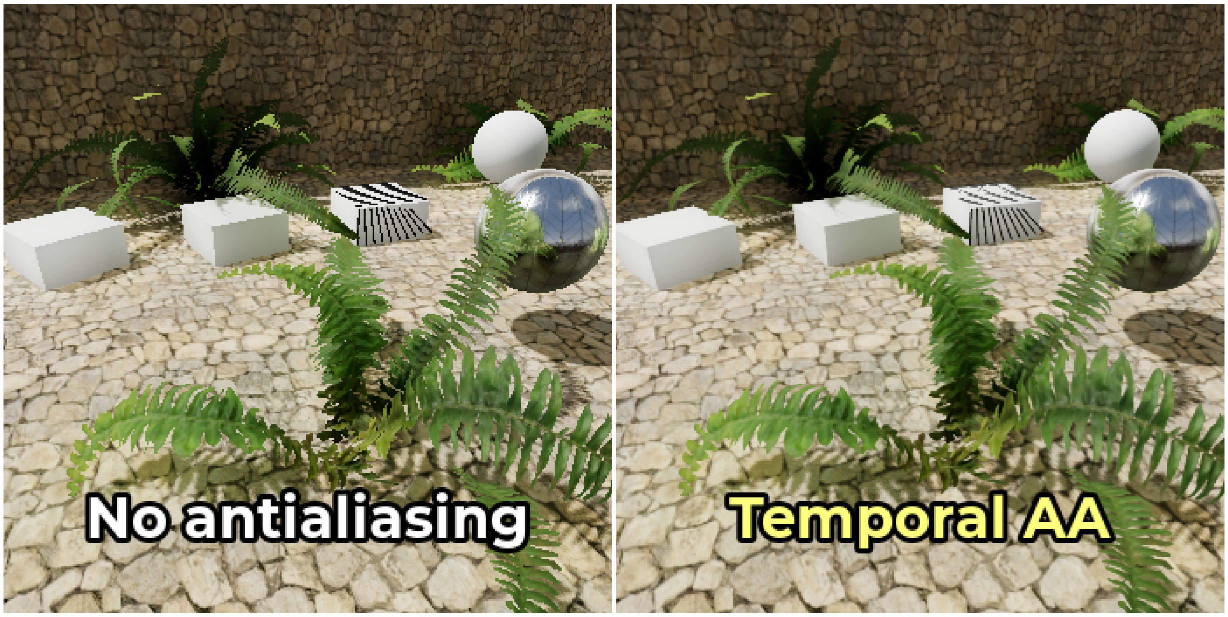
Temporal antialiasing works by *converging* the result of previously rendered frames into a single, high-quality frame. This is a continuous process that works by jittering the position of all vertices in the scene every frame. This jittering is done to capture sub-pixel detail and should be unnoticeable except in extreme situations.

This technique is commonly used in modern games, as it provides the most effective form of antialiasing against specular aliasing and other shader-induced artifacts. TAA also provides full support for transparency antialiasing.

TAA introduces a small amount of blur when enabled in still scenes, but this blurring effect becomes more pronounced when the camera is moving. Another downside of TAA is that it can exhibit *ghosting* artifacts behind moving objects. Rendering at a higher framerate will allow TAA to converge faster, therefore making those ghosting artifacts less visible.

Temporal antialiasing can be enabled in the Project Settings by changing the value of the **Rendering > Anti Aliasing > Quality > Use TAA** setting.

Comparison between no antialiasing (left) and TAA (right):



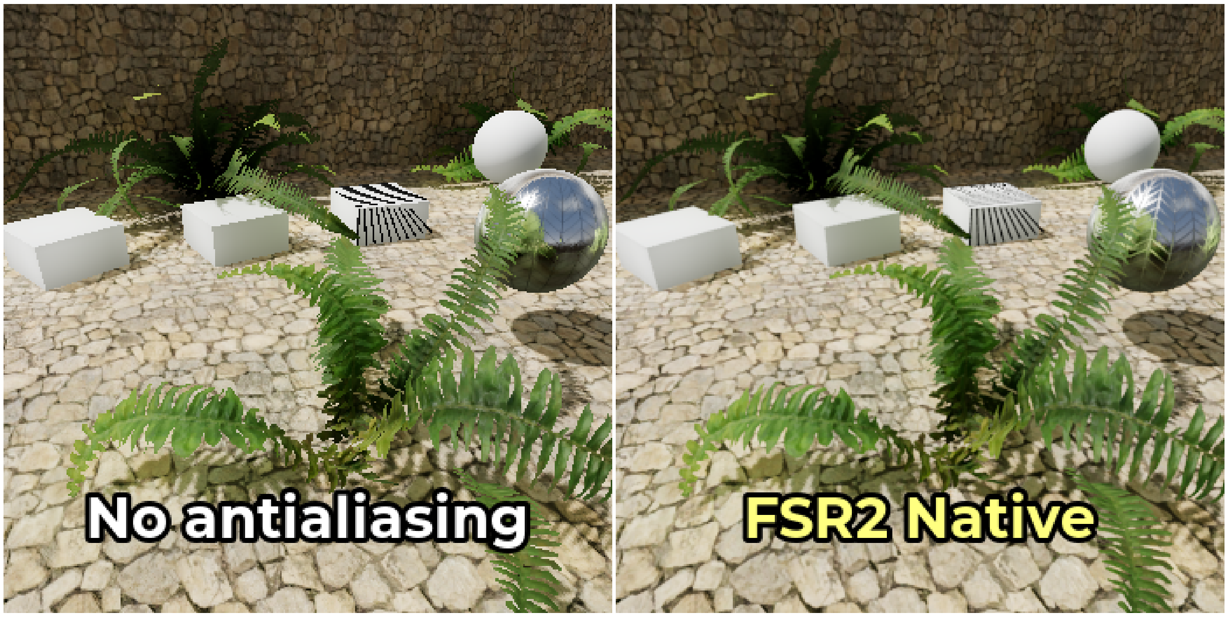
AMD FidelityFX Super Resolution 2.2 (FSR2)

Since Godot 4.2, there is built-in support for AMD FidelityFX Super Resolution [https://www.amd.com/en/products/graphics/technologies/fidelityfx/super-resolution.html] 2.2. This is an upscaling method compatible with all recent GPUs from any vendor. FSR2 is normally designed to improve performance by lowering the internal 3D rendering resolution, then upscaling to the output resolution.

However, unlike FSR1, FSR2 also provides temporal antialiasing. This means FSR2 can be used at native resolution for high-quality antialiasing, with the input resolution being equal to the output resolution. In this situation, enabling FSR2 will actually *decrease* performance, but it will significantly improve rendering quality.

Using FSR2 at native resolution is more demanding than using TAA at native resolution, so its use is only recommended if you have significant GPU headroom. On the bright side, FSR2 provides better antialiasing coverage with less blurriness compared to TAA, especially in motion.

Comparison between no antialiasing (left) and FSR2 at native resolution (right):



Note

By default, the **FSR Sharpness** project setting is set to `0.2` (higher values result in less sharpening). For the purposes of comparison, FSR sharpening has been disabled by setting it to `2.0` on the above screenshot.

Fast approximate antialiasing (FXAA)

This is only available in the Clustered Forward and Forward Mobile backends, not the Compatibility backend.

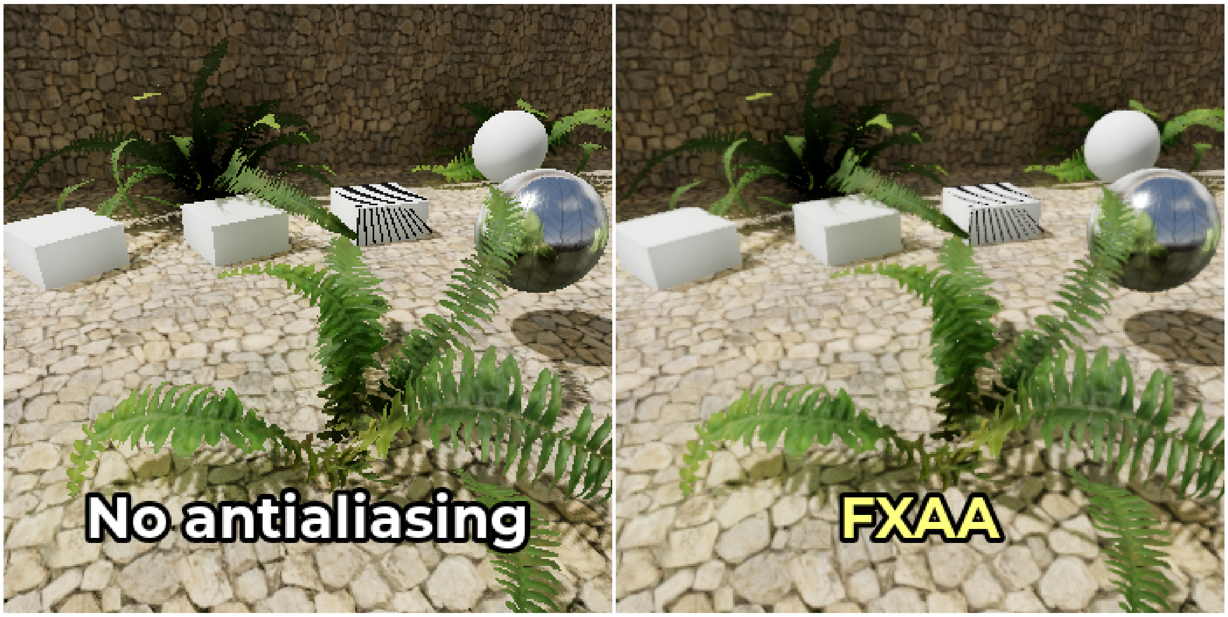
Fast approximate antialiasing is a post-processing antialiasing solution. It is faster to run than any other antialiasing technique and also supports antialiasing transparency. However, since it lacks temporal information, it will not do much against specular aliasing.

This technique is still sometimes used in mobile games. However, on desktop platforms, FXAA generally fell out of fashion in favor of temporal antialiasing, which is much more effective against specular aliasing. Nonetheless, exposing FXAA as an in-game option may still be worthwhile for players with low-end GPUs.

FXAA introduces a moderate amount of blur when enabled (more than TAA when still, but less than TAA when the camera is moving).

FXAA can be enabled in the Project Settings by changing the value of the **Rendering > Anti Aliasing > Quality > Screen Space AA** setting to **FXAA**.

Comparison between no antialiasing (left) and FXAA (right):



Supersample antialiasing (SSAA)

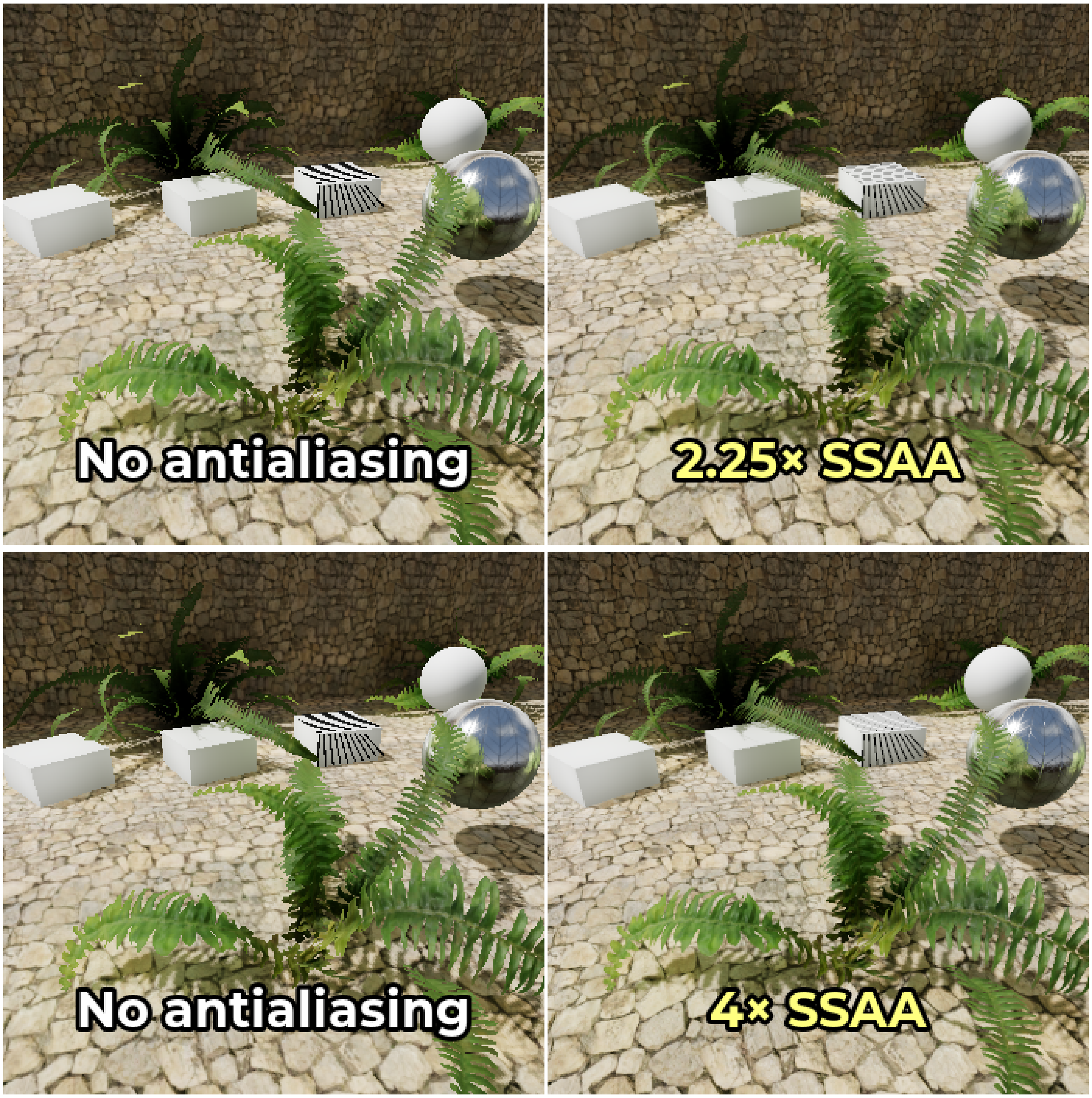
This is only available in the Clustered Forward and Forward Mobile backends, not the Compatibility backend.

Supersampling provides the highest quality of antialiasing possible, but it's also the most expensive. It works by shading every pixel in the scene multiple times. This allows SSAA to antialias edges, transparency *and* specular aliasing at the same time, without introducing potential ghosting artifacts.

The downside of SSAA is its *extremely* high cost. This cost generally makes SSAA difficult to use for game purposes, but you may still find supersampling useful for offline rendering.

Supersample antialiasing is performed by increasing the **Rendering > Scaling 3D > Scale** advanced project setting above `1.0` while ensuring **Rendering > Scaling 3D > Mode** is set to **Bilinear** (the default). Since the scale factor is defined per-axis, a scale factor of `1.5` will result in 2.25× SSAA while a scale factor of `2.0` will result in 4× SSAA. Since Godot uses the hardware's own bilinear filtering to perform the downsampling, the result will look crisper at integer scale factors (namely, `2.0`).

Comparison between no antialiasing (left) and various SSAA levels (right):


No antialiasing
2.25× SSAA
No antialiasing
4× SSAA

Warning

Supersampling also has high video RAM requirements, since it needs to render in the target resolution then *downscale* to the window size. For example, displaying a project in 3840×2160 (4K resolution) with 4× SSAA will require rendering the scene in 7680×4320 (8K resolution), which is 4 times more pixels.

If you are using a high window size such as 4K, you may find that increasing the resolution scale past a certain value will cause a heavy slowdown (or even a crash) due to running out of VRAM.

Screen-space roughness limiter

This is only available in the Clustered Forward and Forward Mobile backends, not the Compatibility backend.

This is not an edge antialiasing method, but it is a way of reducing specular aliasing in 3D.

The screen-space roughness limiter works best on detailed geometry. While it has an effect on roughness map rendering itself, its impact is limited there.

The screen-space roughness limiter is enabled by default; it doesn't require any manual setup. It has a small performance impact, so consider disabling it if your project isn't affected by specular aliasing much.

Texture roughness limiter on import

Like the screen-space roughness limiter, this is not an edge antialiasing method, but it is a way of reducing specular aliasing in 3D.

Roughness limiting on import works by specifying a normal map to use as a guide for limiting roughness. This is done by selecting the roughness map in the FileSystem dock, then going to the Import dock and setting **Roughness > Mode** to the color channel the roughness map is stored in (typically **Green**), then setting the path to the material's normal map. Remember to click **Reimport** at the bottom of the Import dock after setting the path to the normal map.

Since this processing occurs purely on import, it has no performance cost whatsoever. However, its visual impact is limited. Limiting roughness on import only helps reduce specular aliasing within textures, not the aliasing that occurs on geometry edges on detailed meshes.

Which antialiasing technique should I use?

There is no "one size fits all" antialiasing technique. Since antialiasing is often demanding on the GPU or can introduce unwanted blurriness, you'll want to add a setting to allow players to disable antialiasing.

For projects with a photorealistic art direction, TAA is generally the most suitable option. While TAA can introduce ghosting artifacts, there is no other technique that combats specular aliasing as well as TAA does. The screen-space roughness limiter helps a little, but is far less effective against specular aliasing overall. If you have spare GPU power, you can use FSR2 at native resolution for a better-looking form of temporal antialiasing compared to standard TAA.

For projects with a low amount of reflective surfaces (such as a cartoon artstyle), MSAA can work well. MSAA is also a good option if avoiding blurriness and temporal artifacts is important, such as in competitive games.

When targeting low-end platforms such as mobile or integrated graphics, FXAA is usually the only viable option. 2× MSAA may be usable in some circumstances, but higher MSAA levels are unlikely to run smoothly on mobile GPUs.

Godot allows using multiple antialiasing techniques at the same time. This is usually unnecessary, but it can provide better visuals on high-end GPUs or for non-real-time rendering. For example, to make moving edges look better when TAA is enabled, you can also enable MSAA at the same time.

Using MultiMeshInstance3D

Introduction

In a normal scenario, you would use a MeshInstance3D node to display a 3D mesh like a human model for the main character, but in some cases, you would like to create multiple instances of the same mesh in a scene. You *could* duplicate the same node multiple times and adjust the transforms manually. This may be a tedious process and the result may look mechanical. Also, this method is not conducive to rapid iterations. MultiMeshInstance3D is one of the possible solutions to this problem.

MultiMeshInstance3D, as the name suggests, creates multiple copies of a MeshInstance over a surface of a specific mesh. An example would be having a tree mesh populate a landscape mesh with trees of random scales and orientations.

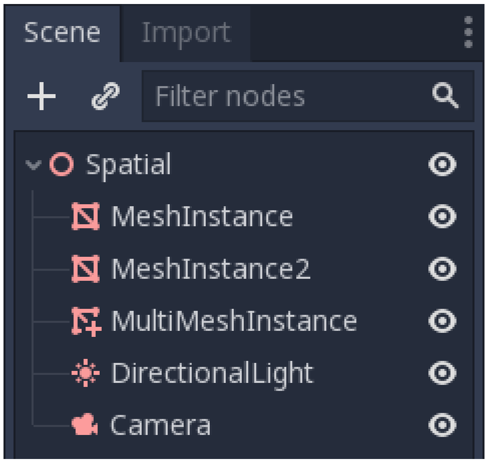
Setting up the nodes

The basic setup requires three nodes: the MultiMeshInstance3D node and two MeshInstance3D nodes.

One node is used as the target, the surface mesh that you want to place multiple meshes on. In the tree example, this would be the landscape.

The other node is used as the source, the mesh that you want to have duplicated. In the tree case, this would be the tree itself.

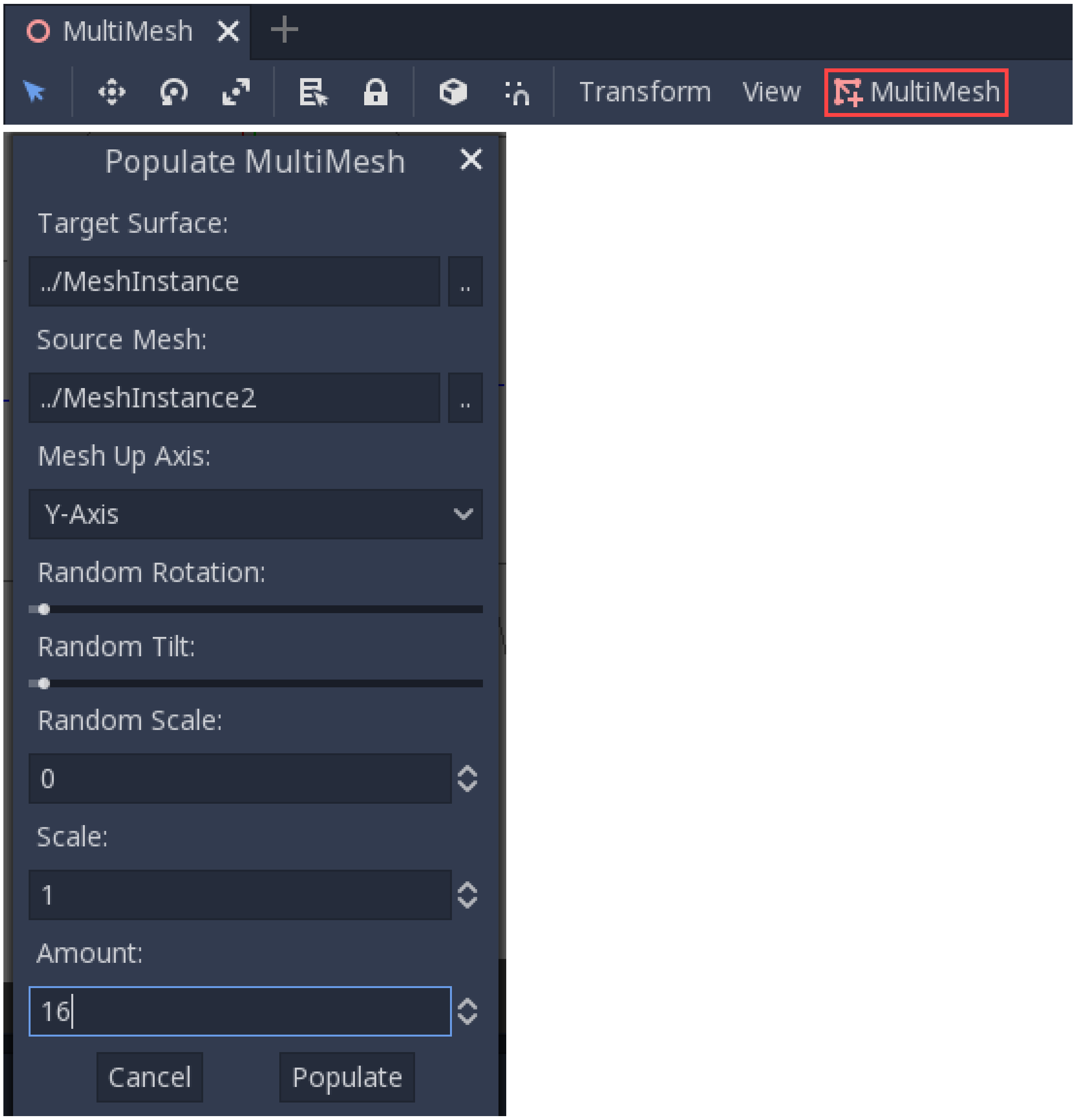
In our example, we would use a Node3D node as the root node of the scene. Your scene tree would look like this:


Scene
Import
Filter nodes
Spatial
MeshInstance
MeshInstance2
MultiMeshInstance
DirectionalLight
Camera

Note

For simplicity's sake, this tutorial uses built-in primitives.

Now you have everything ready. Select the MultiMeshInstance3D node and look at the toolbar, you should see an extra button called `MultiMesh` next to `View`. Click it and select *Populate surface* in the dropdown menu. A new window titled *Populate MultiMesh* will pop up.


MultiMesh
Transform
View
MultiMesh

Populate MultiMesh

Target Surface: ../MeshInstance

Source Mesh: ../MeshInstance2

Mesh Up Axis: Y-Axis

Random Rotation:

Random Tilt:

Random Scale: 0

Scale: 1

Amount: 16

Cancel
Populate

MultiMesh settings

Below are descriptions of the options.

Target Surface

The mesh used as the target surface on which to place copies of your source mesh.

Source Mesh

The mesh you want duplicated on the target surface.

Mesh Up Axis

The axis used as the up axis of the source mesh.

Random Rotation

Randomizing the rotation around the up axis of the source mesh.

Random Tilt

Randomizing the overall rotation of the source mesh.

Random Scale

Randomizing the scale of the source mesh.

Scale

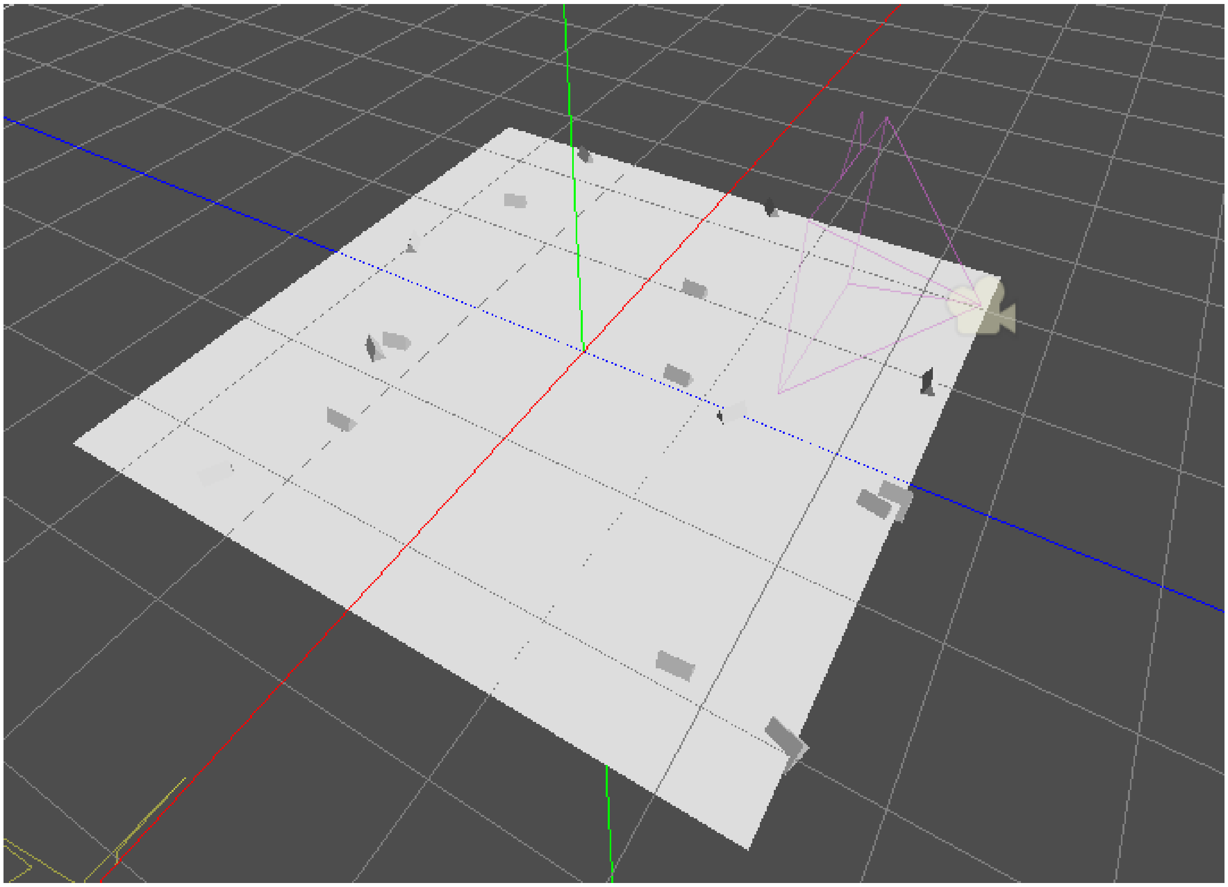
The scale of the source mesh that will be placed over the target surface.

Amount

The amount of mesh instances placed over the target surface.

Select the target surface. In the tree case, this should be the landscape node. The source mesh should be the tree node. Adjust the other parameters according to your preference. Press `Populate` and multiple copies of the source mesh will be placed over the target mesh. If you are satisfied with the result, you can delete the mesh instance used as the source mesh.

The end result should look like this:



To change the result, repeat the previous steps with different parameters.

Mesh level of detail (LOD)

Level of detail (LOD) is one of the most important ways to optimize rendering performance in a 3D project, along with Occlusion culling.

On this page, you'll learn:

- How mesh LOD can improve your 3D project's rendering performance.
- How to set up mesh LOD in Godot.
- How to measure mesh LOD's effectiveness in your project (and alternatives you can explore if it doesn't meet your expectations).

See also

You can see how mesh LOD works in action using the Occlusion Culling and Mesh LOD demo project [https://github.com/godotengine/godot-demo-projects/tree/master/3d/occlusion_culling_mesh_lod].

Introduction

Historically, level of detail in 3D games involved manually authoring meshes with lower geometry density, then configuring the distance thresholds at which these lower-detailed meshes should be drawn. This approach is still used today when increased control is needed.

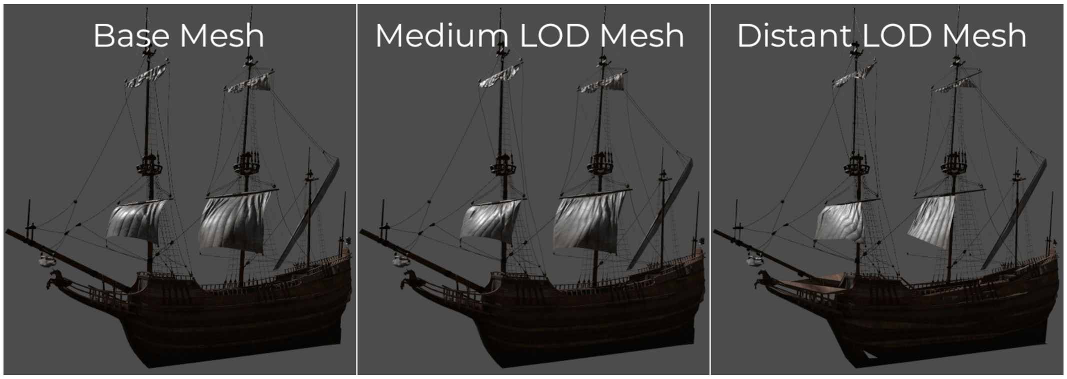
However, in projects that have a large amount of detailed 3D assets, setting up LOD manually can be a very time-consuming process. As a result, automatic mesh decimation and LOD configuration is becoming increasingly popular.

Godot provides a way to automatically generate less detailed meshes for LOD usage on import, then use those LOD meshes when needed automatically. This is completely transparent to the user. The meshoptimizer [https://meshoptimizer.org/] library is used for LOD mesh generation behind the scenes.

Mesh LOD works with any node that draws 3D meshes. This includes MeshInstance3D, MultiMeshInstance3D, GPUParticles3D and CPUParticles3D.

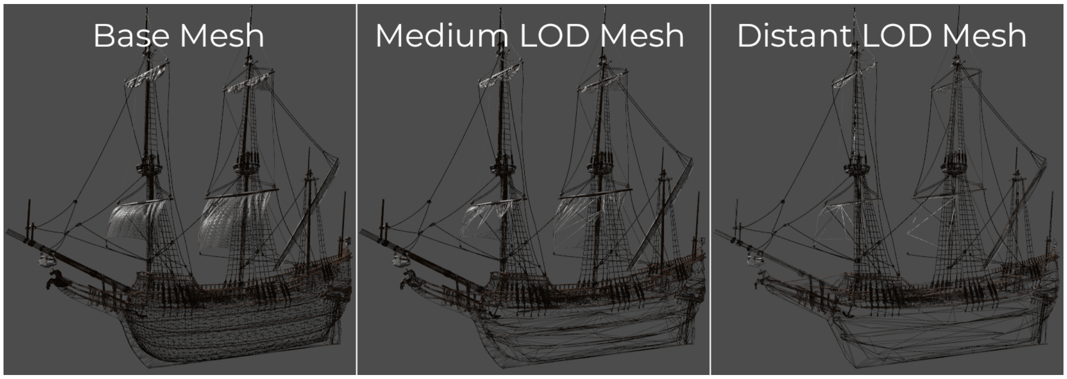
Visual comparison

Here is an example of LOD meshes generated on import. Lower detailed meshes will be used when the camera is far away from the object:



From most detailed (left) to least detailed (right), shaded view

Here's the same image with wireframe rendering to make the decimation easier to see:



From most detailed (left) to least detailed (right), wireframe view

See also

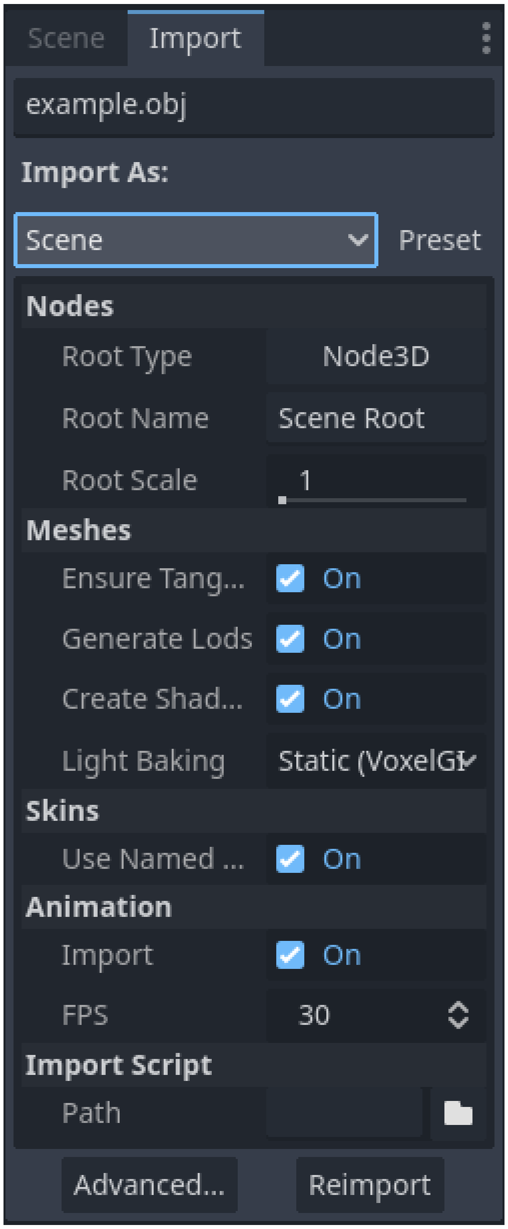
If you need to manually configure level of detail with artist-created meshes, use Visibility ranges (HLOD) instead of automatic mesh LOD.

Generating mesh LOD

By default, mesh LOD generation happens automatically for imported 3D scenes (glTF, .blend, Collada, FBX). Once LOD meshes are generated, they will automatically be used when rendering the scene. You don't need to configure anything manually.

However, mesh LOD generation does **not** automatically happen for imported 3D meshes (OBJ). This is because OBJ files are not imported as full 3D scenes by default, but only as individual mesh resources to load into a MeshInstance3D node (or GPUParticles3D, CPUParticles3D, ...).

To make an OBJ file have mesh LOD generated for it, select it in the FileSystem dock, go to the Import dock, change its **Import As** option to **Scene** then click **Reimport**:



example.obj

Import As: Scene

Nodes

Root Type Node3D

Root Name Scene Root

Root Scale 1

Meshes

Ensure Tang... ☑ On

Generate Lods ☑ On

Create Shad... ☑ On

Light Baking Static (VoxelGI

Skins

Use Named ... ☑ On

Animation

Import ☑ On

FPS 30

Import Script

Path

Changing the import type on an OBJ file in the Import dock

This will require restarting the editor after clicking **Reimport**.

Note

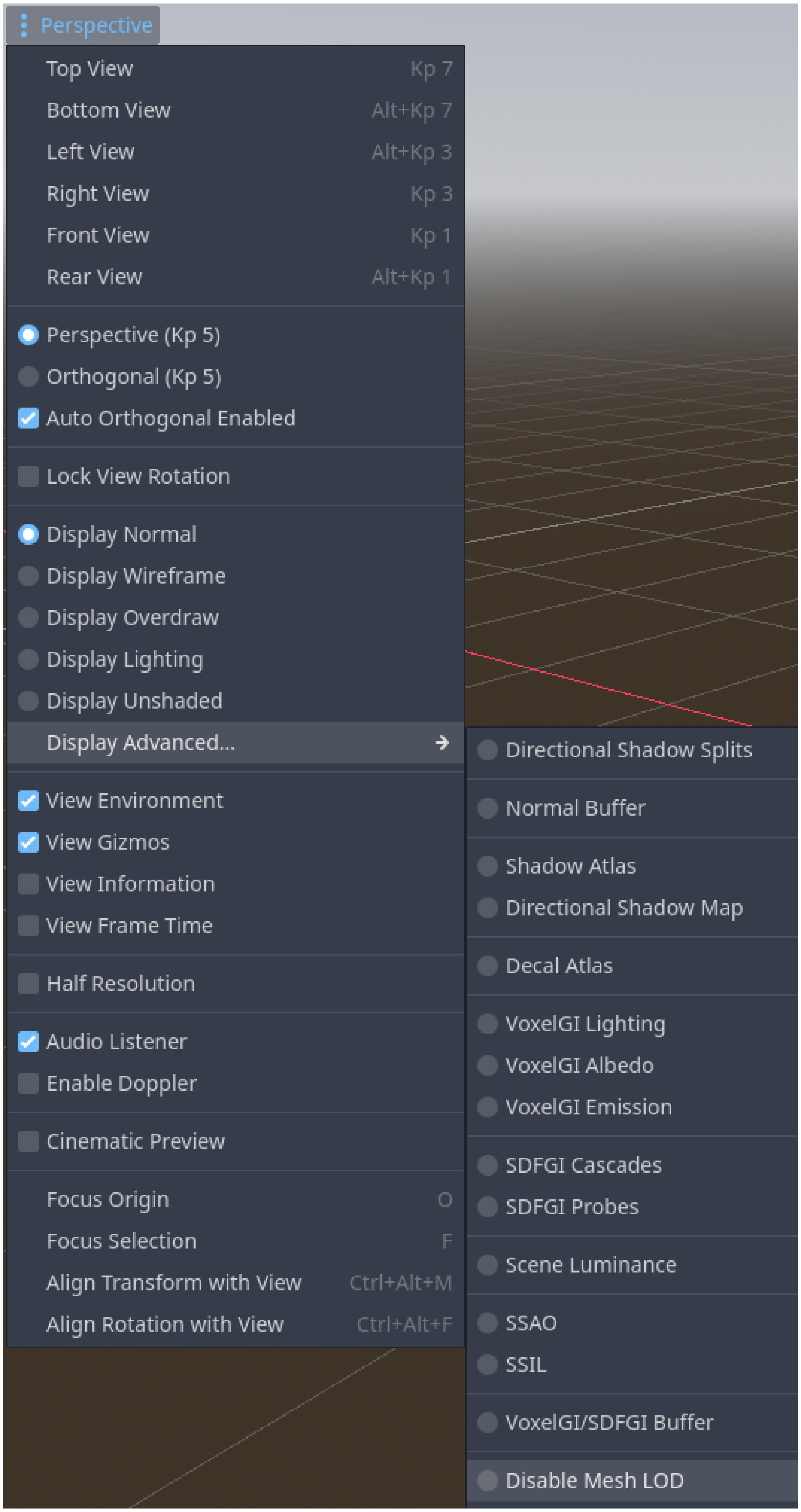
The mesh LOD generation process is not perfect, and may occasionally introduce rendering issues (especially in skinned meshes). Mesh LOD generation can also take a while on complex meshes.

If mesh LOD causes a specific mesh to look broken, you can disable LOD generation for it in the Import dock. This will also speed up resource importing. This can be done globally in the 3D scene's import options, or on a per-mesh basis using the Advanced Import Settings dialog.

See Importing 3D scenes for more information.

Comparing mesh LOD visuals and performance

To disable mesh LOD in the editor for comparison purposes, use the **Disable Mesh LOD** advanced debug draw mode. This can be done using the menu in the top-left corner of the 3D viewport (labeled **Perspective** or **Orthogonal** depending on camera mode):


Perspective
Top View Kp 7
Bottom View Alt+Kp 7
Left View Alt+Kp 3
Right View Kp 3
Front View Kp 1
Rear View Alt+Kp 1
● Perspective (Kp 5)
○ Orthogonal (Kp 5)
☑ Auto Orthogonal Enabled
☐ Lock View Rotation
● Display Normal
○ Display Wireframe
○ Display Overdraw
○ Display Lighting
○ Display Unshaded
Display Advanced...
☑ View Environment
☑ View Gizmos
☐ View Information
☐ View Frame Time
☐ Half Resolution
☑ Audio Listener
☐ Enable Doppler
☐ Cinematic Preview
Focus Origin O
Focus Selection F
Align Transform with View Ctrl+Alt+M
Align Rotation with View Ctrl+Alt+F
○ Directional Shadow Splits
○ Normal Buffer
○ Shadow Atlas
○ Directional Shadow Map
○ Decal Atlas
○ VoxelGI Lighting
○ VoxelGI Albedo
○ VoxelGI Emission
○ SDFGI Cascades
○ SDFGI Probes
○ Scene Luminance
○ SSAO
○ SSIL
○ VoxelGI/SDFGI Buffer
○ Disable Mesh LOD

Disabling mesh LOD in the 3D viewport's top-left menu

Enable **View Frame Time** in the same menu to view FPS in the top-right corner. Also enable **View Information** in the same menu to view the number of primitives (vertices + indices) rendered in the bottom-right corner.

If mesh LOD is working correctly in your scene and your camera is far away enough from the mesh, you should notice the number of drawn primitives decreasing and FPS increasing when mesh LOD is left enabled (unless you are CPU-bottlenecked).

To see mesh LOD decimation in action, change the debug draw mode to **Display Wireframe** in the menu specified above, then adjust the **Rendering > Mesh LOD > LOD Change > Threshold Pixels** project setting.

Configuring mesh LOD performance and quality

You can adjust how aggressive mesh LOD transitions should be in the root viewport by changing the **Rendering > Mesh LOD > LOD Change > Threshold Pixels** project setting. To change this value at run-time, set `mesh_lod_threshold` on the root viewport as follows:

GDScript

```
get_tree().root.mesh_lod_threshold = 4.0
```

C#

```
GetTree().Root.MeshLodThreshold = 4.0f;
```

Each viewport has its own `mesh_lod_threshold` property, which can be set independently from other viewports.

The default mesh LOD threshold of 1 pixel is tuned to look *perceptually* lossless; it provides a significant performance gain with an unnoticeable loss in quality. Higher values will make LOD transitions happen sooner when the camera moves away, resulting in higher performance but lower quality.

If you need to perform per-object adjustments to mesh LOD, you can adjust how aggressive LOD transitions should be by adjusting the **LOD Bias** property on any node that inherits from GeometryInstance3D. Values *above* `1.0` will make LOD transitions happen later than usual (resulting in higher quality but lower performance). Values *below* `1.0` will make LOD transitions happen sooner than usual (resulting in lower quality but higher performance).

Additionally, ReflectionProbe nodes have their own **Mesh LOD Threshold** property that can be adjusted to improve rendering performance when the reflection probe updates. This is especially important for ReflectionProbes that use the **Always** update mode.

Note

When rendering the scene, mesh LOD selection uses a screen-space metric. This means it automatically takes camera field of view and viewport resolution into account. Higher camera FOV and lower viewport resolutions will make LOD selection more aggressive; the engine will display heavily decimated models earlier when the camera moves away.

As a result, unlike Visibility ranges (HLOD), you don't need to do anything specific in your project to take camera FOV and viewport resolution into account.

Using mesh LOD with MultiMesh and particles

For LOD selection, the point of the node's AABB that is the closest to the camera is used as a basis. This applies to any kind of mesh LOD (including for individual MeshInstance3D)s, but this has some implications for nodes that display multiple meshes at once, such as MultiMeshInstance3D, GPUParticles3D and GPUParticles3D. Most importantly, this means that all instances will be drawn with the same LOD level at a given time.

If you are noticing incorrect LOD selection with GPUParticles3D, make sure the node's visibility AABB is configured by selecting the GPUParticles3D node and using **GPUParticles3D > Generate AABB** at the top of the 3D viewport.

If you have instances in a MultiMesh that are far away from each other, they should be placed in a separate MultiMeshInstance3D node. Doing so will also improve rendering performance, as frustum and occlusion culling will be able to cull individual nodes (while they can't cull individual instances in a MultiMesh).

Visibility ranges (HLOD)

Along with Mesh level of detail (LOD) and Occlusion culling, visibility ranges are another tool to improve performance in large, complex 3D scenes.

On this page, you'll learn:

- What visibility ranges can do and which scenarios they are useful in.
- How to set up visibility ranges (manual LOD) in Godot.
- How to tune visibility ranges for best performance and quality.

See also

If you only need meshes to become less detailed over distance but don't have manually authored LOD meshes, consider relying on automatic Mesh level of detail (LOD) instead.

Note that automatic mesh LOD and visibility ranges can be used at the same time, even on the same mesh.

How it works

Visibility ranges can be used with any node that inherits from GeometryInstance3D. This means they can be used not only with MeshInstance3D and MultiMeshInstance3D for artist-controlled HLOD, but also GPUParticles3D, CPUParticles3D, Label3D, Sprite3D, AnimatedSprite3D and CSGShape3D.

Since visibility ranges are configured on a per-node basis, this makes it possible to use different node types as part of a LOD system. For example, you could display a MeshInstance3D representing a tree when up close, and replace it with a Sprite3D impostor in the distance to improve performance.

The benefit of HLOD over a traditional LOD system is its hierarchical nature. A single larger mesh can replace several smaller meshes, so that the number of draw calls can be reduced at a distance, but culling opportunities can be preserved when up close. For example, you can have a group of houses that uses individual MeshInstance3D nodes (one for each house) when up close, but turns into a single MeshInstance3D that represents a less detailed group of houses (or use a MultiMeshInstance3D).

Lastly, visibility ranges can also be used to fade certain objects entirely when the camera gets too close or too far. This can be used for gameplay purposes, but also to reduce visual clutter. For example, Label3D nodes can be faded using visibility ranges when they're too far away to be readable or relevant to the player.

Setting up visibility range

This is a quick-start guide for configuring a basic LOD system. After following this guide, this LOD system will display a SphereMesh when up close and a BoxMesh when the camera is far away enough. A small hysteresis margin is also configured via the **Begin Margin** and **End Margin** properties. This prevents LODs from popping back and forth too quickly when the camera is moving at the "edge" of the LOD transition.

The visibility range properties can be found in the **Visibility Range** section of the GeometryInstance3D inspector after selecting the MeshInstance3D Node.

- Add a Node3D node that will be used to group the two MeshInstance3D nodes together.

- Add a first MeshInstance3D node as a child of the Node3D. Assign a new SphereMesh to its Mesh property.
- Set the first MeshInstance3D's visibility range **End** to `10.0` and **End Margin** to `1.0`.
- Add a second MeshInstance3D node as a child of the Node3D. Assign a new BoxMesh to its Mesh property.
- Set the second MeshInstance3D's visibility range **Begin** to `10.0` and **Begin Margin** to `1.0`.
- Move the camera away and back towards the object. Notice how the object will transition from a sphere to a box as the camera moves away.

Visibility range properties

In the inspector of any node that inherits from GeometryInstance3D, you can adjust the following properties in the GeometryInstance3D's **Visibility Range** section:

- **Begin:** The instance will be hidden when the camera is closer to the instance's *origin* than this value (in 3D units).
- **Begin Margin:** The hysteresis or alpha fade transition distance to use for the close-up transition (in 3D units). The behavior of this property depends on **Fade Mode**.
- **End:** The instance will be hidden when the camera is further away from the instance's *origin* than this value (in 3D units).
- **End Margin:** The hysteresis or alpha fade transition distance to use for the far-away transition (in 3D units). The behavior of this property depends on **Fade Mode**.
- **Fade Mode:** Controls how the transition between LOD levels should be performed. See below for details.

Fade mode

Note

The fade mode chosen only has a visible impact if either **Visibility Range > Begin Margin** or **Visibility Range > End Margin** is greater than `0.0`.

In the inspector's **Visibility Range** section, there are 3 fade modes to choose from:

- **Disabled:** Uses hysteresis to switch between LOD levels instantly. This prevents situations where LOD levels are switched back and forth quickly when the player moves forward and then backward at the LOD transition point. The hysteresis distance is determined by **Visibility Range > Begin Margin** and **Visibility Range > End Margin**. This mode provides the best performance as it doesn't force rendering to become transparent during the fade transition.
- **Self:** Uses alpha blending to smoothly fade between LOD levels. The node will fade-out itself when reaching the limits of its own visibility range. The fade transition distance is determined by **Visibility Range > Begin Margin** and **Visibility Range > End Margin**. This mode forces transparent rendering on the object during its fade transition, so it has a performance impact.
- **Dependencies:** Uses alpha blending to smoothly fade between LOD levels. The node will fade-in its dependencies when reaching the limits of its own visibility range. The fade transition distance is determined by **Visibility Range > Begin Margin** and **Visibility Range > End Margin**. This mode forces transparent rendering on the object during its fade transition, so it has a performance impact. This mode is intended for hierarchical LOD systems using Visibility parent. It acts the same as **Self** if visibility ranges are used to perform non-hierarchical LOD.

Visibility parent

The **Visibility Parent** property makes it easier to set up HLOD. It allows automatically hiding child nodes if its parent is visible given its current visibility range properties.

Note

The target of **Visibility Parent** *must* inherit from GeometryInstance3D.

Despite its name, the **Visibility Parent** property *can* point to a node that is not a parent of the node in the scene tree. However, it is impossible to point **Visibility Parent** towards a child node, as this creates a dependency cycle which is not supported. You will get an error message in the Output panel if a dependency cycle occurs.

Given the following scene tree (where all nodes inherit from GeometryInstance3D):

```
└─ BatchOfHouses
    ├─ House1
    ├─ House2
    ├─ House3
    └─ House4
```

In this example, *BatchOfHouses* is a large mesh designed to represent all child nodes when viewed at a distance. *House1* to *House4* are smaller MeshInstance3Ds representing individual houses. To configure HLOD in this example, we only need to configure two things:

- Set **Visibility Range Begin** to a number greater than *0.0* so that *BatchOfHouses* only appears when far away enough from the camera. Below this distance, we want *House1* to *House4* to be displayed instead.

- On *House1* to *House4*, assign the **Visibility Parent** property to *BatchOfHouses*.

This makes it easier to perform further adjustments, as you don't need to adjust the **Visibility Range Begin** of *BatchOfHouses* and **Visibility Range End** of *House1* to *House4*.

Fade mode is automatically handled by the **Visibility Parent** property, so that the child nodes only become hidden once the parent node is fully faded out. This is done to minimize visible pop-in. Depending on your HLOD setup, you may want to try both the **Self** and **Dependencies** fade modes.

Note

Nodes hidden via the **Visible** property are essentially removed from the visibility dependency tree, so dependent instances will not take the hidden node or its ancestors into account.

In practice, this means that if the target of the **Visibility Parent** node is hidden by setting its **Visible** property to `false`, the node will not be hidden according to the **Visibility Range Begin** value specified in the visibility parent.

Configuration tips

Use simpler materials at a distance to improve performance

One way to further improve performance is to use simpler materials for distant LOD meshes. While using LOD meshes will reduce the number of vertices that need to be rendered, the per-pixel shading load for materials remains identical. However, per-pixel shading load is regularly a bottleneck on the GPU in complex 3D scenes. One way to reduce this shading load on the GPU is to use simpler materials when they don't make much of a visual difference.

Performance gains when doing so should be carefully measured, as increasing the number of *unique* materials in a scene has a performance cost on its own. Still, using simpler materials for distant LOD meshes can still result in a net performance gain as a result of the fewer per-pixel calculations required.

For example, on the materials used by distant LOD meshes, you can disable expensive material features such as:

- Normal Map (especially on mobile platforms)
- Rim
- Clearcoat
- Anisotropy
- Height
- Subsurface Scattering
- Back Lighting
- Refraction
- Proximity Fade

Use dithering for LOD transitions

Godot currently only supports alpha-based fading for visibility ranges. You can however use dithering instead by using several different materials for different LOD levels.

There are two advantages to using dithering over alpha blending for LOD transitions:

- Higher performance, as dithering transparency is faster to render compared to alpha blending.
- No visual glitches due to transparency sorting issues during LOD transitions.

The downside of dithering is that a "noisy" pattern is visible during LOD fade transitions. This may not be as noticeable at higher viewport resolutions or when temporal antialiasing is enabled.

Also, as distance fade in BaseMaterial3D only supports fading up close *or* fading when far away, this setup is best used with only two LODs as part of the setup.

- Ensure **Begin Margin** and **End Margin** is set to `0.0` on both MeshInstance3D nodes, as hysteresis or alpha fade are not desired here.
- On both MeshInstance3D nodes, *decrease* **Begin** by the desired fade transition distance and *increase* **End** by the same distance. This is required for the dithering transition to actually be visible.
- On the MeshInstance3D that is displayed up close, edit its material in the inspector. Set its **Distance Fade** mode to **Object Dither**. Set **Min Distance** to the same value as the visibility range **End**. Set **Max Distance** to the same value *minus* the fade transition distance.
- On the MeshInstance3D that is displayed far away, edit its material in the inspector. Set its **Distance Fade** mode to **Object Dither**. Set **Min Distance** to the same value as the visibility range **Begin**. Set **Max Distance** to the same value *plus* the fade transition distance.

Occlusion culling

In a 3D rendering engine, **occlusion culling** is the process of performing hidden geometry removal.

On this page, you'll learn:

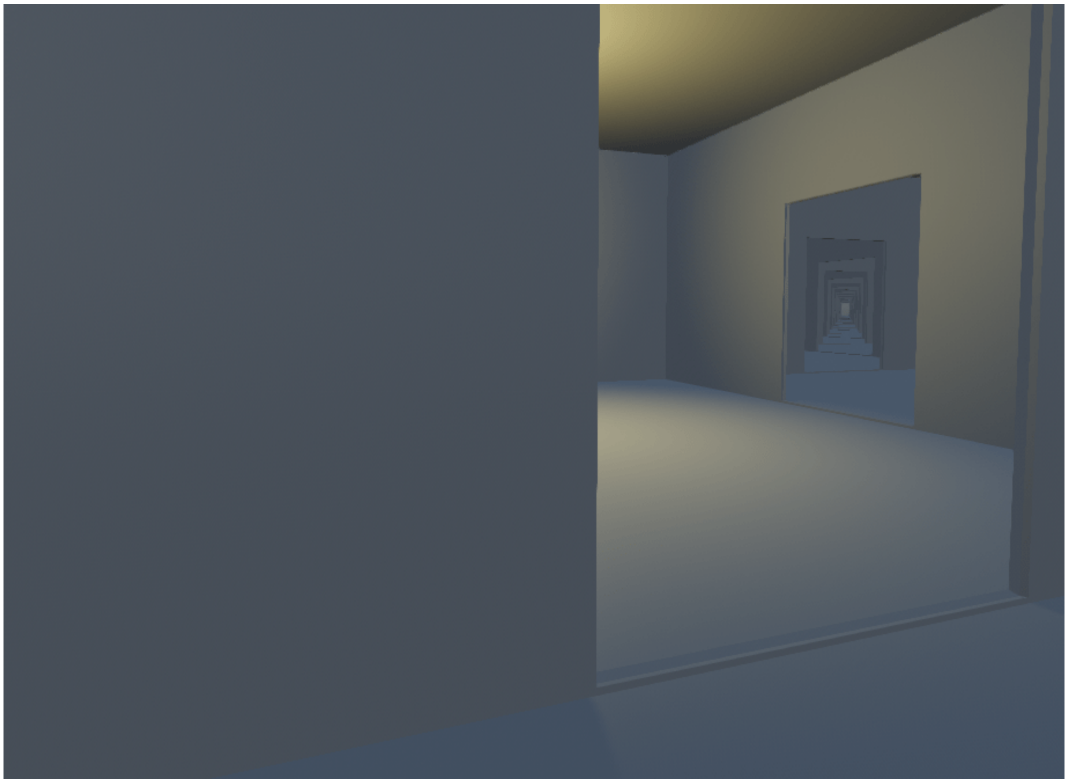
- What are the advantages and pitfalls of occlusion culling.
- How to set up occlusion culling in Godot.
- Troubleshooting common issues with occlusion culling.

See also

You can see how occlusion culling works in action using the Occlusion Culling and Mesh LOD demo project [https://github.com/godotengine/godot-demo-projects/tree/master/3d/occlusion_culling_mesh_lod].

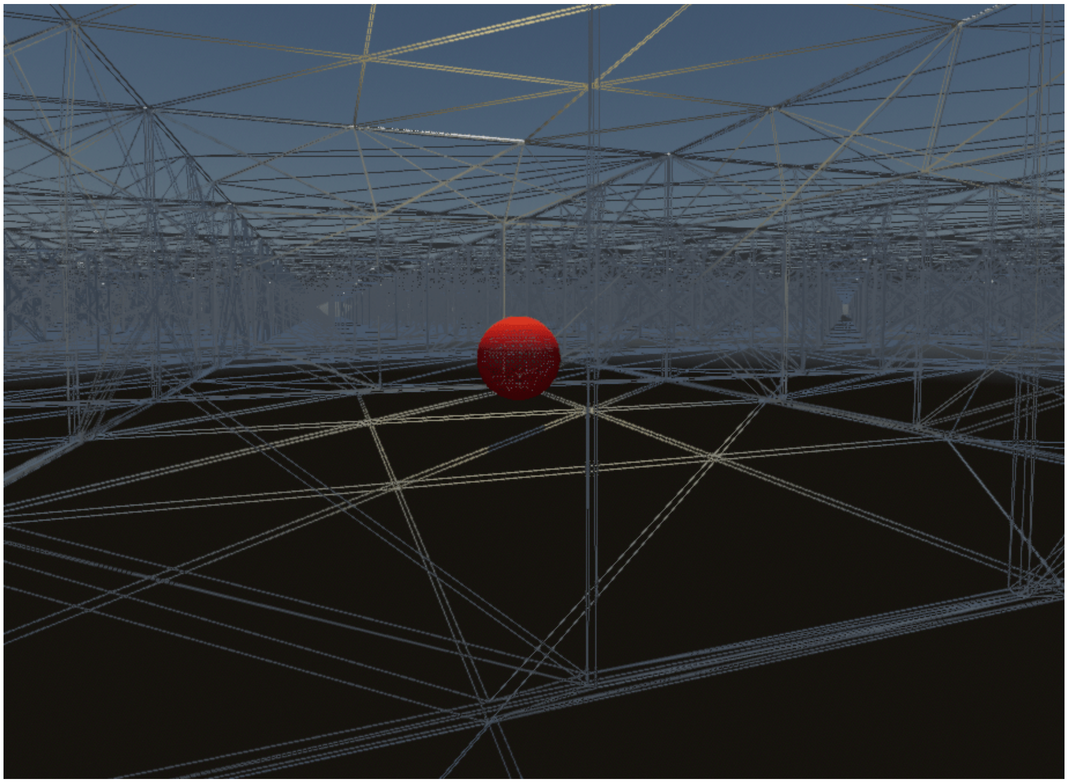
Why use occlusion culling

In this example scene with hundreds of rooms stacked next to each other, a dynamic object (red sphere) is hidden behind the wall in the lit room (on the left of the door):



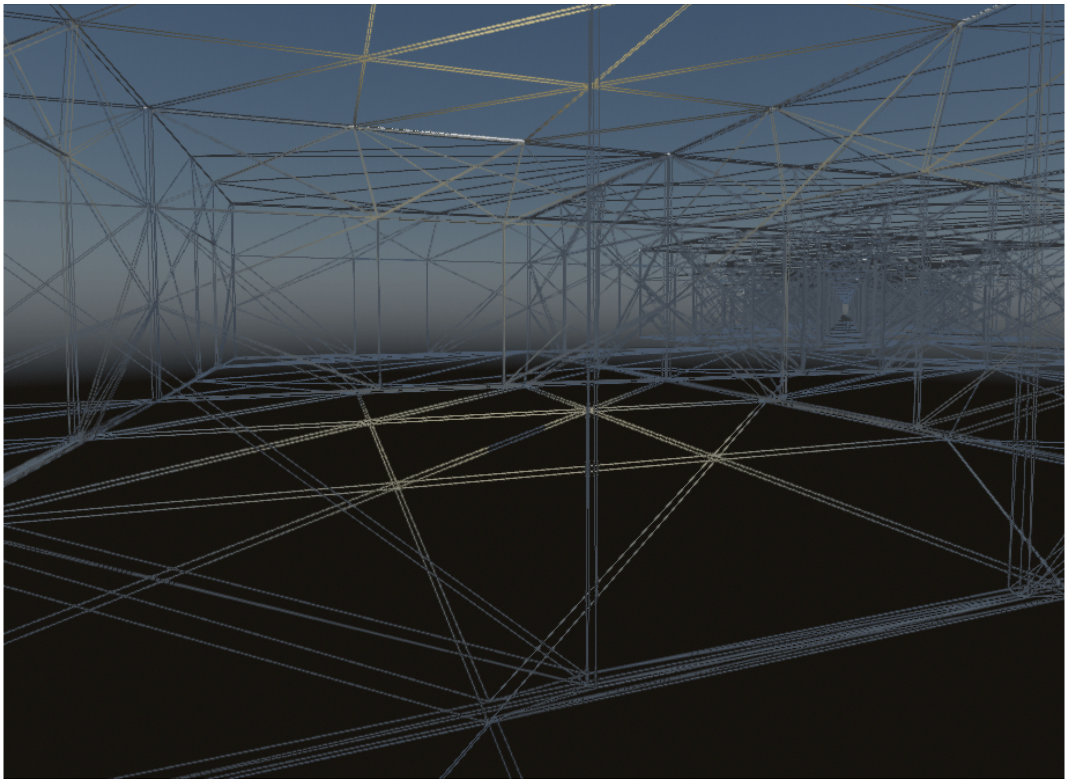
Example scene with an occlusion culling-friendly layout

With occlusion culling disabled, all the rooms behind the lit room have to be rendered. The dynamic object also has to be rendered:



Example scene with occlusion culling **disabled** (wireframe)

With occlusion culling enabled, only the rooms that are actually visible have to be rendered. The dynamic object is also occluded by the wall, and therefore no longer has to be rendered:



Example scene with occlusion culling **enabled** (wireframe)

Since the engine has less work to do (fewer vertices to render and fewer draw calls), performance will increase as long as there are enough occlusion culling opportunities in the scene. This means occlusion culling is most effective in indoor scenes, preferably with many smaller rooms instead of fewer larger rooms. Combine this with Mesh level of detail (LOD) and Visibility ranges (HLOD) to further improve performance gains.

Note

When using the Clustered Forward rendering backend, the engine already performs a *depth prepass*. This consists in rendering a depth-only version of the scene before rendering the scene's actual materials. This is used to ensure each opaque pixel is only shaded once, reducing the cost of overdraw significantly.

The greatest performance benefits can be observed when using the Forward Mobile rendering backend, as it does not feature a depth prepass for performance reasons. As a result, occlusion culling will actively decrease shading overdraw with that rendering backend.

Nonetheless, even when using a depth prepass, there is still a noticeable benefit to occlusion culling in complex 3D scenes. However, in scenes with few occlusion culling opportunities, occlusion culling may not be worth the added setup and CPU usage.

How occlusion culling works in Godot

Note

"occluder" refers to the shape blocking the view, while "occludee" refers to the object being hidden.

In Godot, occlusion culling works by rasterizing the scene's occluder geometry to a low-resolution buffer on the CPU. This is done using the software raytracing library Embree [https://github.com/embree/embree].

The engine then uses this low-resolution buffer to test the occludee's AABB against the occluder shapes. The occludee's AABB must be *fully occluded* by the occluder shape to be culled.

As a result, smaller objects are more likely to be effectively culled than larger objects. Larger occluders (such as walls) also tend to be much more effective than smaller ones (such as decoration props).

Setting up occlusion culling

The first step to using occlusion culling is to enable the **Rendering > **Occlusion Culling > Use Occlusion Culling** project setting. (Make sure the **Advanced** toggle is enabled in the Project Settings dialog to be able to see it.)

This project setting applies immediately, so you don't need to restart the editor.

After enabling the project setting, you still need to create some occluders. For performance reasons, the engine doesn't automatically use all visible geometry as a basis for occlusion culling. Instead, the engine requires a simplified representation of the scene with only static objects to be baked.

There are two ways to set up occluders in a scene:

Automatically baking occluders (recommended)

Note

Only MeshInstance3D nodes are currently taken into account in the *occluder* baking process. MultiMeshInstance3D, GPUParticles3D, CPUParticles3D and CSG nodes are **not** taken into account when baking occluders. If you wish those to be treated as occluders, you have to manually create occluder shapes that (roughly) match their geometry.

This restriction does not apply to *occludees*. Any node type that inherits from GeometryInstance3D can be occluded.

After enabling the occlusion culling project setting mentioned above, add an OccluderInstance3D node to the scene containing your 3D level.

Select the OccluderInstance3D node, then click **Bake Occluders** at the top of the 3D editor viewport. After baking, the OccluderInstance3D node will contain an Occluder3D resource that stores a simplified version of your level's geometry. This occluder geometry appears as purple wireframe lines in the 3D view (as long as **View Gizmos** is enabled in the **Perspective** menu). This geometry is then used to provide occlusion culling for both static and dynamic occludees.

After baking, you may notice that your dynamic objects (such as the player, enemies, etc…) are included in the baked mesh. To prevent this, set the **Bake > Cull Mask** property on the OccluderInstance3D to exclude certain visual layers from being baked.

For example, you can disable layer 2 on the cull mask, then configure your dynamic objects' MeshInstance3D nodes to be located on the visual layer 2 (instead of layer 1). To do so, select the MeshInstance3D node in question, then on the **VisualInstance3D > Layers** property, uncheck layer 1 then check layer 2. After configuring both cull mask and layers, bake occluders again by following the above process.

Manually placing occluders

This approach is more suited for specialized use cases, such as creating occlusion for MultiMeshInstance3D setups or CSG nodes (due to the aforementioned limitation).

After enabling the occlusion culling project setting mentioned above, add an OccluderInstance3D node to the scene containing your 3D level. Select the OccluderInstance3D node, then choose an occluder type to add in the **Occluder** property:

- QuadOccluder3D (a single plane)
- BoxOccluder3D (a cuboid)
- SphereOccluder3D (a sphere-shaped occluder)
- PolygonOccluder3D (a 2D polygon with as many points as you want)

There is also ArrayOccluder3D, whose points can't be modified in the editor but can be useful for procedural generation from a script.

Previewing occlusion culling

You can enable a debug draw mode to preview what the occlusion culling is actually "seeing". In the top-left corner of the 3D editor viewport, click the **Perspective** button (or **Orthogonal** depending on your current camera mode), then choose **Display Advanced... > Occlusion Culling Buffer**. This will display the low-resolution buffer that is used by the engine for occlusion culling.

In the same menu, you can also enable **View Information** and **View Frame Time** to view the number of draw calls and rendered primitives (vertices + indices) in the bottom-right corner, along with the number of frames per second rendered in the top-right corner.

If you toggle occlusion culling in the project settings while this information is displayed, you can see how much occlusion culling improves performance in your scene. Note that the performance benefit highly depends on the 3D editor camera's view angle, as occlusion culling is only effective if there are occluders in front of the camera.

To toggle occlusion culling at run-time, set `use_occlusion_culling` on the root viewport as follows:

GDScript

```
get_tree().root.use_occlusion_culling = true
```

C#

```
GetTree().Root.UseOcclusionCulling = true;
```

Toggling occlusion culling at run-time is useful to compare performance on a running project.

Performance considerations

Design your levels to take advantage of occlusion culling

This is the most important guideline. A good level design is not just about what the gameplay demands; it should also be built with occlusion in mind.

For indoor environments, add opaque walls to "break" the line of sight at regular intervals and ensure not too much of the scene can be seen at once.

For large open scenes, use a pyramid-like structure for the terrain's elevation when possible. This provides the greatest culling opportunities compared to any other terrain shape.

Avoid moving OccluderInstance3D nodes during gameplay

This includes moving the parents of OccluderInstance3D nodes, as this will cause the nodes themselves to move in global space, therefore requiring the BVH to be rebuilt.

Toggling an OccluderInstance3D's visibility (or one of its parents' visibility) is not as expensive, as the update only needs to happen once (rather than continuously).

For example, if you have a sliding or rotating door, you can make the OccluderInstance3D node not be a child of the door itself (so that the occluder never moves), but you can hide the OccluderInstance3D visibility once the door starts opening. You can then reshow the OccluderInstance3D once the door is fully closed.

If you absolutely have to move an OccluderInstance3D node during gameplay, use a primitive Occluder3D shape for it instead of a complex baked shape.

Use the simplest possible occluder shapes

If you notice low performance or stuttering in complex 3D scenes, it may mean that the CPU is overloaded as a result of rendering detailed occluders. Select the OccluderInstance3D node, increase the **Bake > Simplification** property then bake occluders again.

Remember to keep the simplification value reasonable. Values that are too high for the level's geometry may cause incorrect occlusion culling to occur, as in My occludee is being culled when it shouldn't be.

If this still doesn't lead to low enough CPU usage, you can try adjusting the **Rendering > Occlusion Culling > BVH Build Quality** project setting and/or decreasing **Rendering > Occlusion Culling > Occlusion Rays Per Thread**. You'll need to enable the **Advanced** toggle in the Project Settings dialog to see those settings.

Troubleshooting

My occludee isn't being culled when it should be

On the occluder side:

First, double-check that the **Bake > Cull Mask** property in the OccluderInstance3D is set to allow baking the meshes you'd like. The visibility layer of the MeshInstance3D nodes must be present within the cull mask for the mesh to be included in the bake.

Also note that occluder baking only takes meshes with *opaque* materials into account. Surfaces will *transparent* materials will **not** be included in the bake, even if the texture applied on them is fully opaque.

Lastly, remember that MultiMeshInstance3D, GPUParticles3D, CPUParticles3D and CSG nodes are **not** taken into account when baking occluders. As a workaround, you can add OccluderInstance3D nodes for those manually.

On the occludee side:

Make sure **Extra Cull Margin** is set as low as possible (it should usually be `0.0`), and that **Ignore Occlusion Culling** is disabled in the object's GeometryInstance3D section.

Also, check the AABB's size (which is represented by an orange box when selecting the node). This axis-aligned bounding box must be *fully* occluded by the occluder shapes for the occludee to be hidden.

My occludee is being culled when it shouldn't be

The most likely cause for this is that objects that were included in the occluder bake have been moved after baking occluders. For instance, this can occur when moving your level geometry around or rearranging its layout. To fix this, select the OccluderInstance3D node and bake occluders again.

This can also happen because dynamic objects were included in the bake, even though they shouldn't be. Use the occlusion culling debug draw mode to look for occluder shapes that shouldn't be present, then adjust the bake cull mask accordingly.

The last possible cause for this is overly aggressive mesh simplification during the occluder baking process. Select the OccluderInstance3D node, decrease the **Bake > Simplification** property then bake occluders again.

As a last resort, you can enable the **Ignore Occlusion Culling** property on the occludee. This will negate the performance improvements of occlusion culling for that object, but it makes sense to do this for objects that will never be culled (such as a first-person view model).

Resolution scaling

Why use resolution scaling?

With the ever-increasing rendering complexity of modern games, rendering at native resolution isn't always viable anymore, especially on lower-end GPUs.

Resolution scaling is one of the most direct ways to influence the GPU requirements of a scene. In scenes that are bottlenecked by the GPU (rather than by the CPU), decreasing the resolution scale can improve performance significantly. Resolution scaling is particularly important on mobile GPUs where performance and power budgets are limited.

While resolution scaling is an important tool to have, remember that resolution scaling is not intended to be a replacement for decreasing graphics settings on lower-end hardware. Consider exposing both resolution scale and graphics settings in your in-game menus.

See also

You can compare resolution scaling modes and factors in action using the 3D Antialiasing demo project [https://github.com/godotengine/godot-demo-projects/tree/master/3d/antialiasing].

Note

Resolution scaling is currently not available for 2D rendering, but it can be simulated using the `viewport` stretch mode. See Multiple resolutions for more information.

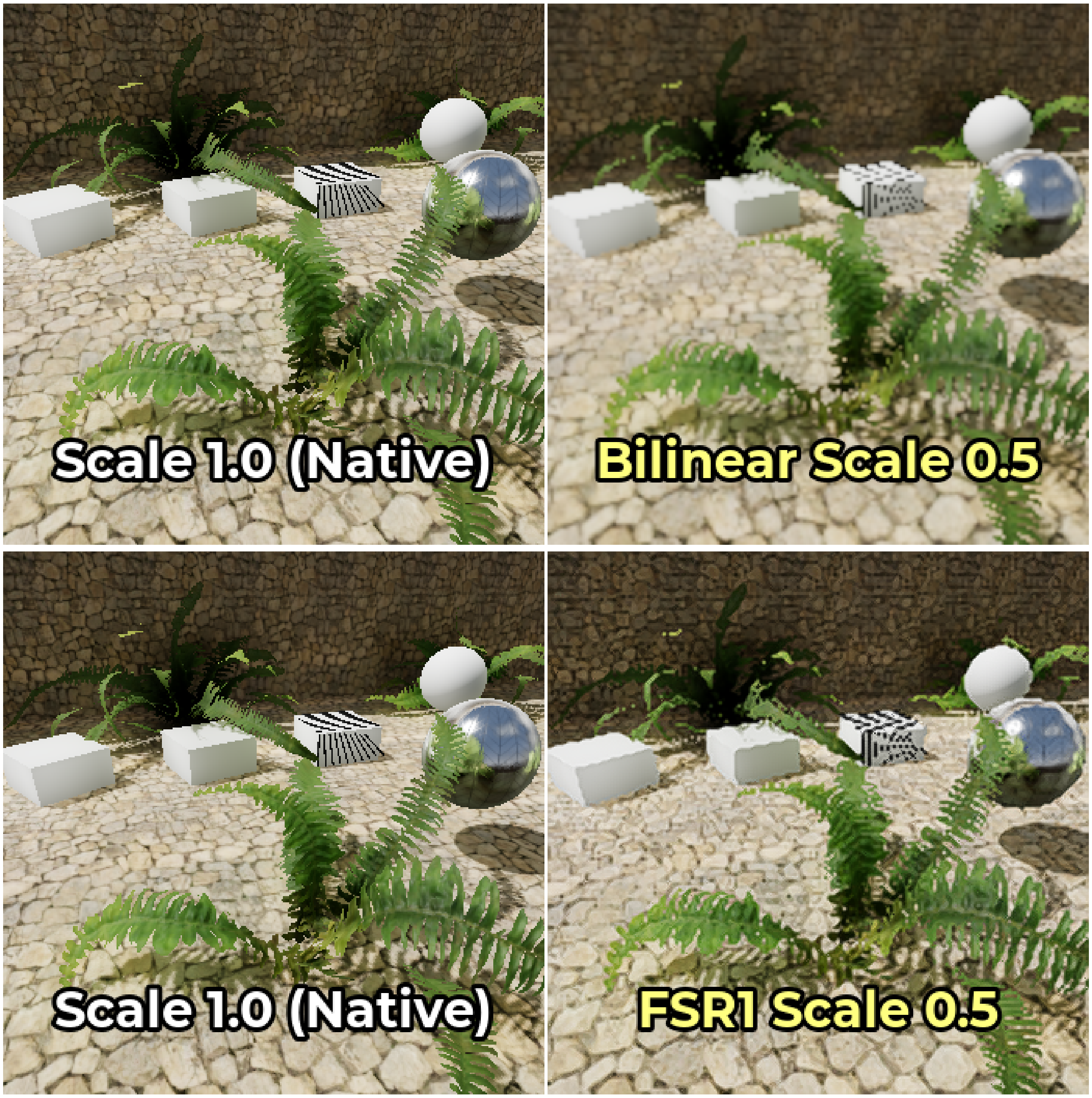
Resolution scaling options

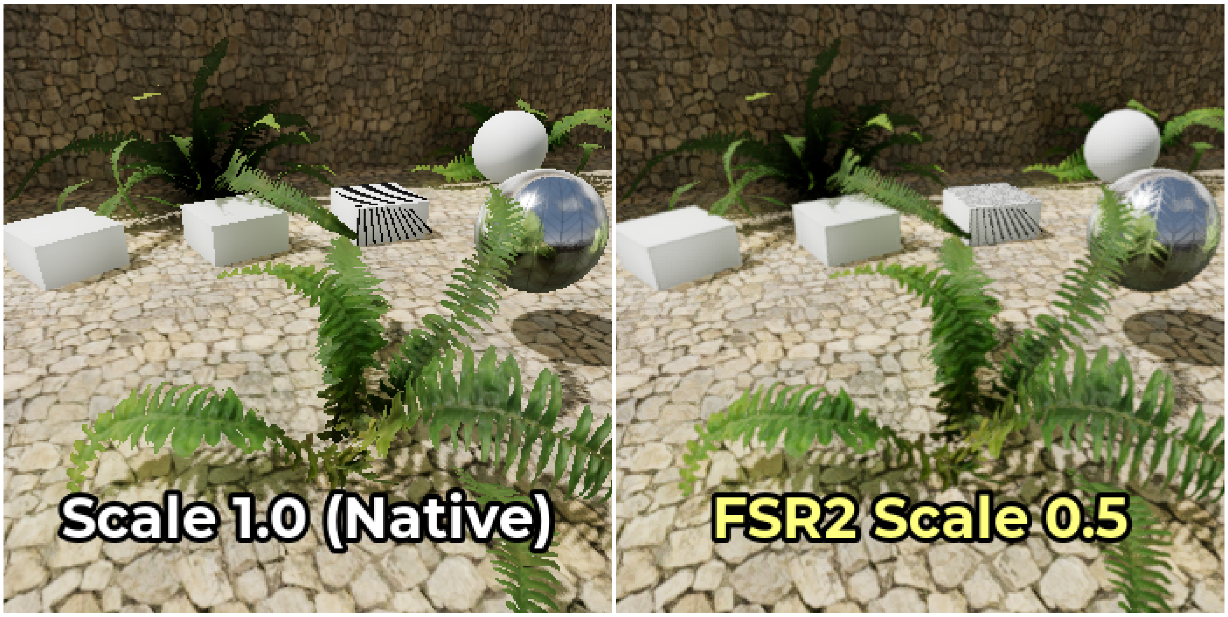
In the advanced Project Settings' **Rendering > Scaling 3D** section, you can find several options for 3D resolution scaling:

Scaling mode

- **Bilinear:** Standard bilinear filtering (default).
- **FSR 1.0:** AMD FidelityFX Super Resolution 1.0 [https://gpuopen.com/fidelityfx-superresolution/]. Slower, but higher quality compared to bilinear scaling. On very slow GPUs, the cost of FSR1 may be too expensive to be worth using it over bilinear scaling.
- **FSR 2.2:** AMD FidelityFX Super Resolution 2.2 (since Godot 4.2). Slowest, but even higher quality compared to FSR1 and bilinear scaling. On slow GPUs, the cost of FSR2 may be too expensive to be worth using it over bilinear scaling or FSR1. To match FSR2 performance with FSR1, you need to use a lower resolution scale factor.

Here are comparison images between native resolution, bilinear scaling with 50% resolution scale, FSR1, and FSR2 scaling with 50% resolution scale:

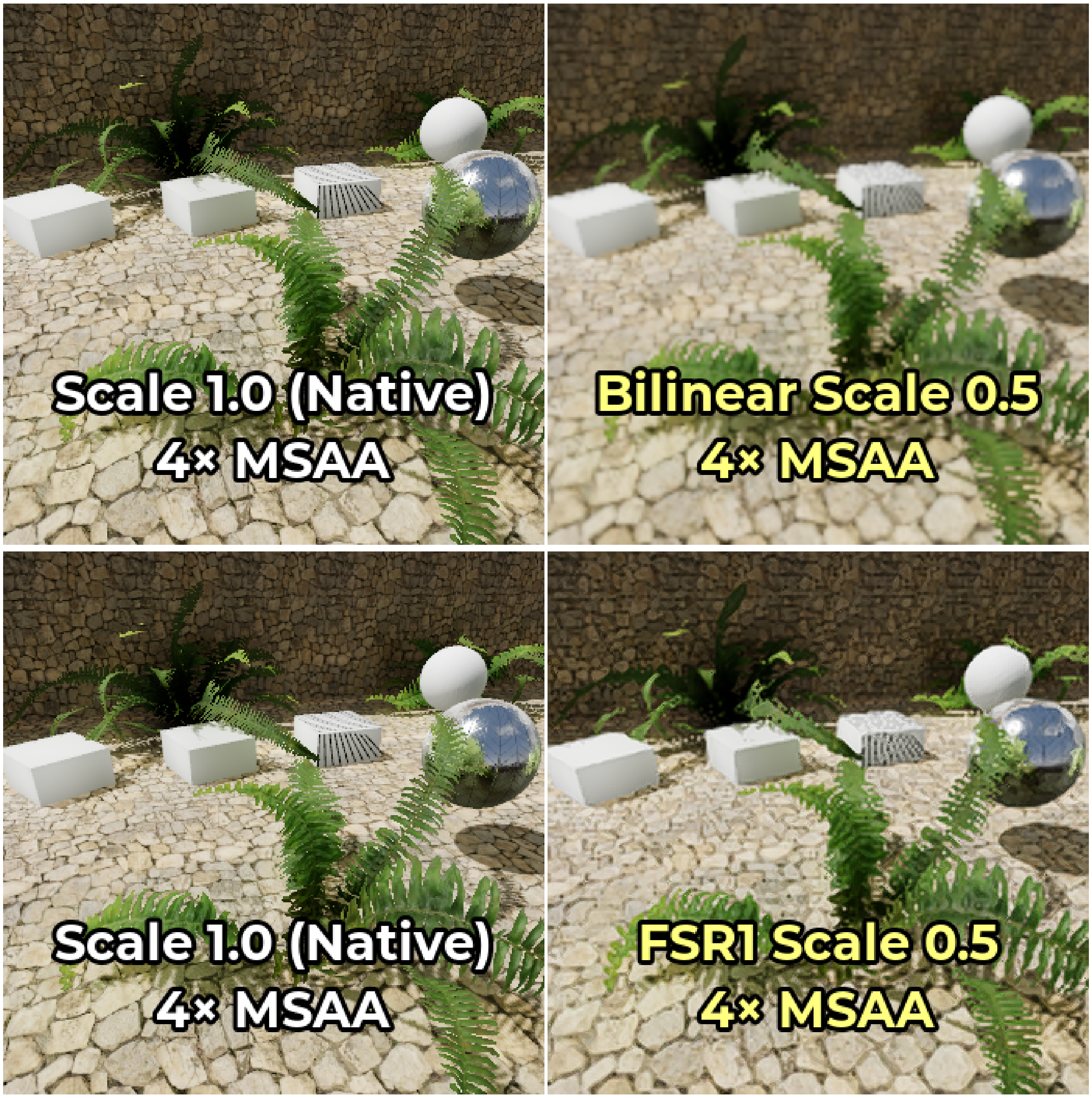

Scale 1.0 (Native)
Bilinear Scale 0.5
Scale 1.0 (Native)
FSR1 Scale 0.5

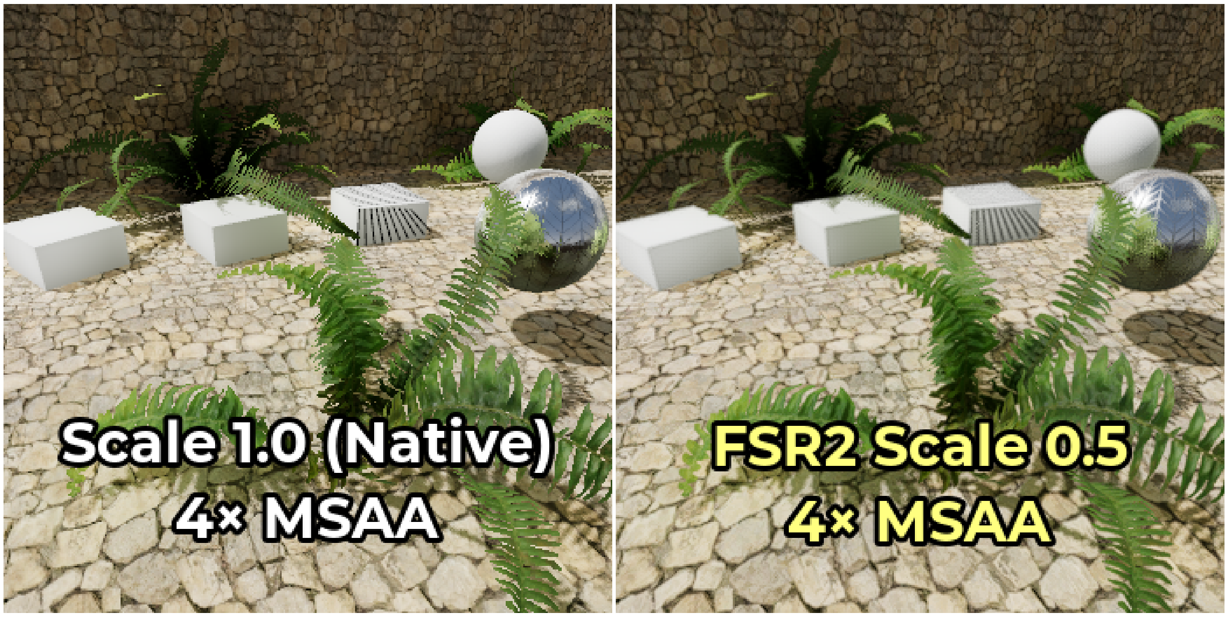


FSR1 upscaling works best when coupled with another form of antialiasing. Temporal antialiasing (TAA) or multisample antialiasing (MSAA) should preferably be used in this case, as FXAA does not add temporal information and introduces more blurring to the image.

On the other hand, FSR2 provides its own temporal antialiasing. This means you don't need to enable other antialiasing methods for the resulting image to look smooth. The **Use TAA** project setting is ignored when FSR2 is used as the 3D scaling method, since FSR2's temporal antialiasing takes priority.

Here's the same comparison, but with 4× MSAA enabled on all images:


Scale 1.0 (Native)
4× MSAA
Bilinear Scale 0.5
4× MSAA
Scale 1.0 (Native)
4× MSAA
FSR1 Scale 0.5
4× MSAA



Notice how the edge upscaling of FSR1 becomes much more convincing once 4× MSAA is enabled. However, FSR2 doesn't benefit much from enabling MSAA since it already performs temporal antialiasing.

Rendering scale

The **Rendering > Scaling 3D > Scale** setting adjusts the resolution scale. `1.0` represents the full resolution scale, with the 3D rendering resolution matching the 2D rendering resolution. Resolution scales *below* `1.0` can be used to speed up rendering, at the cost of a blurrier final image and more aliasing.

The rendering scale can be adjusted at run-time by changing the `scaling_3d_scale` property on a Viewport node.

Resolution scales *above* `1.0` can be used for supersample antialiasing (SSAA). This will provide antialiasing at a *very* high performance cost, and is **not recommended** for most use cases. See 3D antialiasing for more information.

The tables below list common screen resolutions, the resulting 3D rendering resolution and the number of megapixels that need to be rendered each frame depending on the rendering scale option. Rows are sorted from fastest to slowest in each table.

Note

The resolution scale is defined on a **per-axis** basis. For example, this means that halving the resolution scale factor will reduce the number of rendered megapixels per frame by a factor of 4, not 2. Therefore, very low or very high resolution scale factors can have a greater performance impact than expected.

1920×1080 (Full HD)

Resolution scale factor	3D rendering resolution	Megapixels rendered per frame
`0.50`	960×540	0.52 MPix
`0.67`	1286×723	0.93 MPix
`0.75`	1440×810	1.17 MPix
`0.85`	1632×918	1.50 MPix
`1.00` **(native)**	**1920×1080**	**2.07 MPix**

Resolution scale factor	3D rendering resolution	Megapixels rendered per frame
`1.33` (supersampling)	2553×1436	3.67 MPix
`1.50` (supersampling)	2880×1620	4.67 MPix
`2.00` (supersampling)	3840×2160	8.29 MPix

2560×1440 (QHD)

Resolution scale factor	3D rendering resolution	Megapixels rendered per frame
`0.50`	1280×720	0.92 MPix
`0.67`	1715×964	1.65 MPix
`0.75`	1920×1080	2.07 MPix
`0.85`	2176×1224	2.66 MPix
`1.00` **(native)**	**2560×1440**	**3.69 MPix**

Resolution scale factor	3D rendering resolution	Megapixels rendered per frame
`1.33` (supersampling)	3404×1915	6.52 MPix
`1.50` (supersampling)	3840×2160	8.29 MPix
`2.00` (supersampling)	5120×2880	14.75 MPix

3840×2160 (Ultra HD "4K")

Resolution scale factor	3D rendering resolution	Megapixels rendered per frame
`0.50`	1920×1080	2.07 MPix
`0.67`	2572×1447	3.72 MPix
`0.75`	2880×1620	4.67 MPix
`0.85`	3264×1836	5.99 MPix
`1.00` **(native)**	**3840×2160**	**8.29 MPix**

Resolution scale factor	3D rendering resolution	Megapixels rendered per frame
`1.33` (supersampling)	5107×2872	14.67 MPix
`1.50` (supersampling)	5760×3240	18.66 MPix
`2.00` (supersampling)	7680×4320	33.18 MPix

FSR Sharpness

When using the FSR1 or FSR2 scaling modes, the sharpness can be controlled using the **Rendering > Scaling 3D > FSR Sharpness** advanced project setting.

The intensity is inverted compared to most other sharpness sliders: *lower* values will result in a sharper final image, while *higher* values will *reduce* the impact of the sharpening filter. `0.0` is the sharpest, while `2.0` is the least sharp. The default value of `0.2` provides a balance between preserving the original image's sharpness and avoiding additional aliasing due to oversharpening.

Note

If you wish to use sharpening when rendering at native resolution, Godot currently doesn't allow using the sharpening component of FSR1 (RCAS) independently from the upscaling component (EASU).

As a workaround, you can set the 3D rendering scale to `0.99`, set the scaling mode to **FSR 1.0** then adjust FSR sharpness as needed. This allows using FSR1 while rendering at a near-native resolution.

Alternatively, you can set the scaling mode to **FSR 2.2** with the 3D rendering scale set to `1.0` if you have enough GPU headroom. This also provides high-quality temporal antialiasing. The **FSR Sharpness** setting remains functional in this case.

Mipmap bias

Godot automatically uses a negative texture mipmap bias when the 3D resolution scale is set below `1.0`. This allows for better preservation of texture detail at the cost of a grainy appearance on detailed textures.

The texture LOD bias currently affects both 2D and 3D rendering in the same way. However, keep in mind it only has an effect on textures with mipmaps enabled. Textures used in 2D don't have mipmaps enabled by default, which means only 3D rendering is affected unless you enabled mipmaps on 2D textures in the Import dock.

The formula used to determine the texture mipmap bias is: `log2f(min(scaling_3d_scale, 1.0)) + custom_texture_mipmap_bias`

To counteract the blurriness added by some antialiasing methods, Godot also adds a `-0.25` offset when FXAA is enabled, and a `-0.5` offset when TAA is enabled. If both are enabled at the same time, a `-0.75` offset is used. This mipmap bias offset is applied *before* the resolution scaling offset, so it does not change depending on resolution scale.

The texture LOD bias can manually be changed by adjusting the **Rendering > Textures > Default Filters > Texture Mipmap Bias** advanced project setting. It can also be changed at run-time on Viewports by adjusting the `texture_mipmap_bias` property.

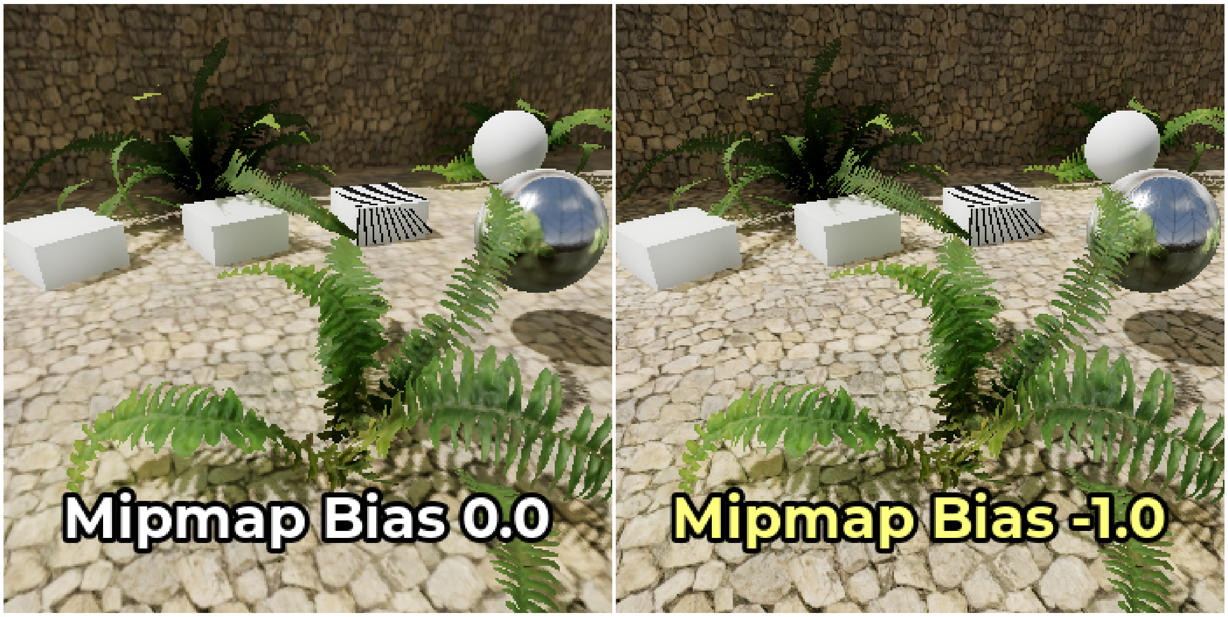
Warning

Adjusting the mipmap LOD bias manually can be useful in certain scenarios, but this should be done carefully to prevent the final image from looking grainy in motion.

Negative mipmap LOD bias can also decrease performance due to higher-resolution mips having to be sampled further away. Recommended values for a manual offset are between `-0.5` and `0.0`.

Positive mipmap LOD bias will make mipmapped textures appear blurrier than intended. This may improve performance slightly, but is otherwise not recommended as the loss in visual quality is usually not worth the performance gain.

The example below shows an extreme case, with a mipmap LOD bias of `-1.0` and anisotropic filtering disabled to make the difference more noticeable:



Troubleshooting

Performance does not increase much when decreasing resolution scale

If performance doesn't increase much when decreasing resolution scale to a value like `0.5`, it likely means the performance bottleneck is elsewhere in your scene. For example, your scene could have too many draw calls, causing a CPU bottleneck to occur. Likewise, you may have too many graphics effects enabled for your GPU to handle (such as SDFGI, SSAO or SSR).

See the Performance tutorials for more information.

Variable rate shading

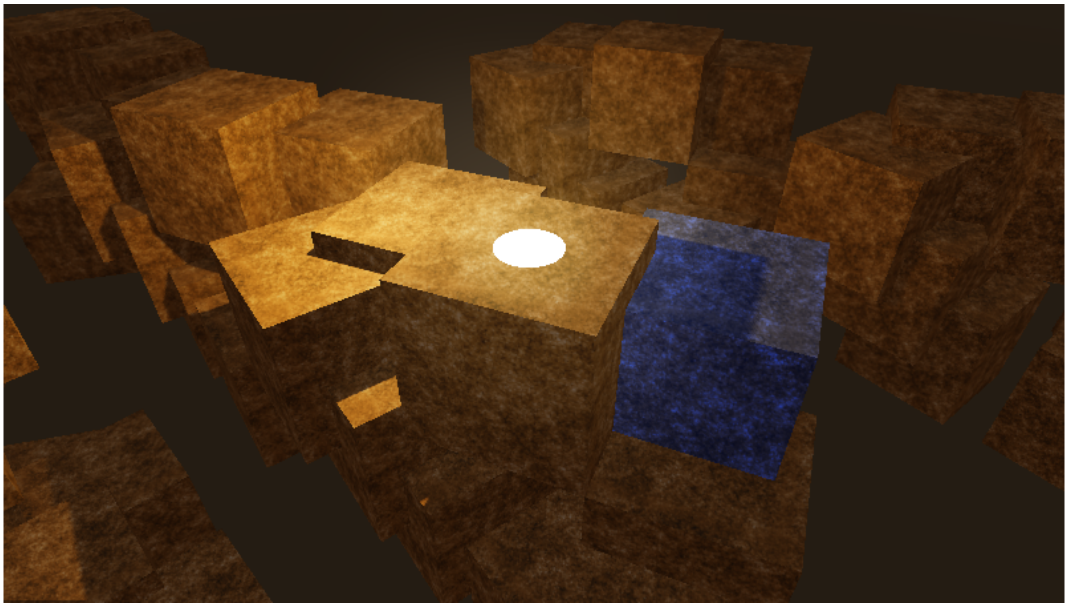
What is variable rate shading?

In modern 3D rendering engines, shaders are much more complex compared to before. The advent of physically-based rendering, real-time global illumination and screen-space effects has increased the number of *per-pixel* shading that must be performed to render each frame. Additionally, screen resolutions also have increased a lot, with 1440p and 4K now being common target resolutions. As a result, the total shading cost in scene rendering usually represents a significant amount of the time taken to render each frame.

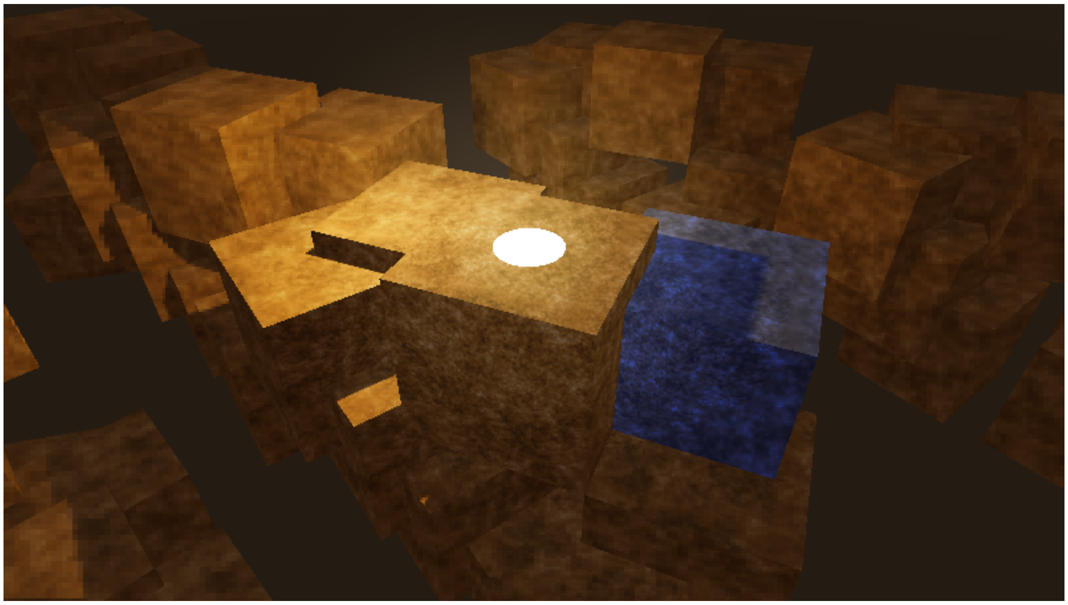
Variable rate shading (VRS) is a method of decreasing this shading cost by reducing the resolution of *per-pixel* shading (also called *fragment* shading), while keeping the original resolution for rendering geometry. This means geometry edges remain as sharp as they would without VRS. VRS can be combined with any 3D antialiasing technique (MSAA, FXAA, TAA, SSAA).

VRS allows specifying the shading quality in a local manner, which makes it possible to have certain parts of the viewport receive more detailed shading than others. This is particularly useful in virtual reality (VR) to achieve *foveated rendering*, where the center of the viewport is more detailed than the edges.

Here's a scene rendered with rate shading disabled then enabled, using the density map linked at the bottom of this page:

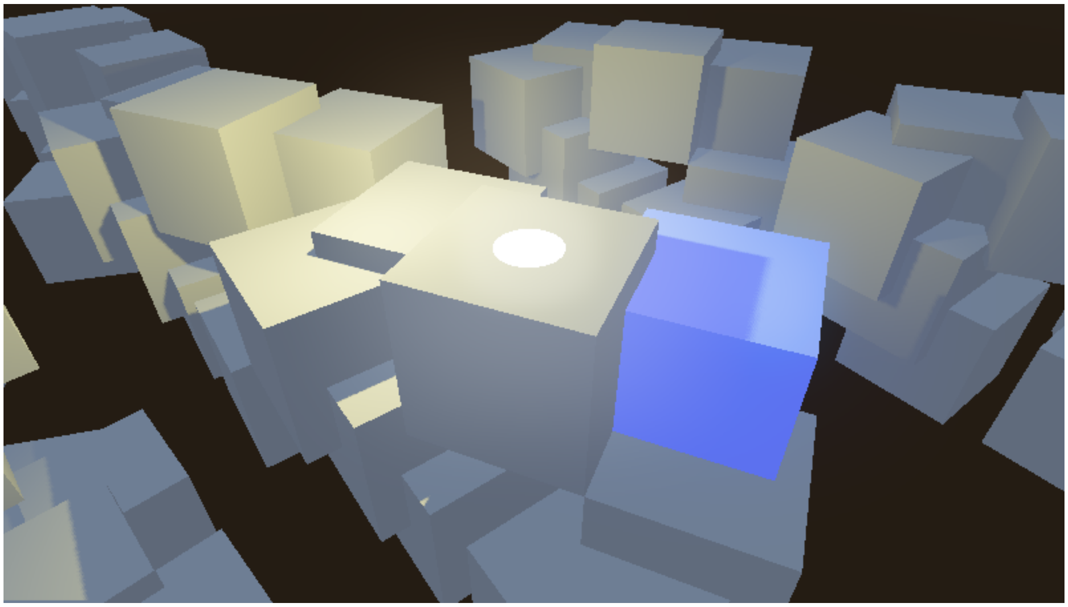


Variable rate shading disabled in textured scene

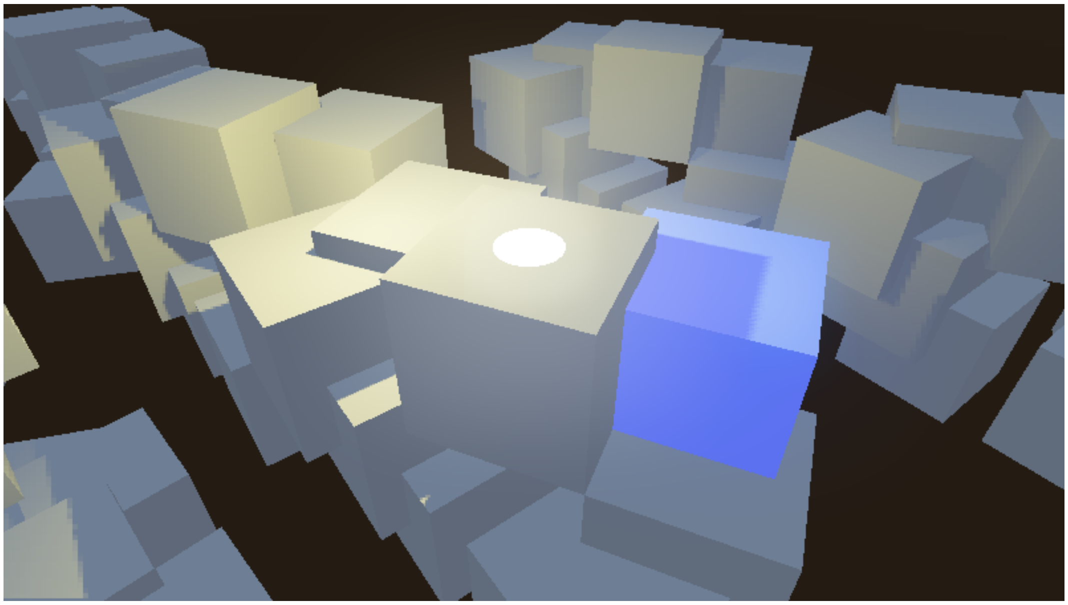


Variable rate shading enabled in textured scene (lower quality but higher performance)

When used in scenes with low-frequency detail (such as scenes with a stylized/low-poly aesthetic), it's possible to achieve similar performance gains but with less reduction in visual quality:



Variable rate shading disabled in untextured scene



Variable rate shading enabled in untextured scene (lower quality but higher performance)

Hardware support

Variable rate shading is only supported on specific GPUs:

Desktop:

- NVIDIA Turing and newer (including GTX 1600 series)
- AMD RDNA2 and newer (both integrated and dedicated GPUs – including Steam Deck)
- Intel Arc Alchemist and newer **(dedicated GPUs only)**
  - Intel integrated graphics do not support variable rate shading.

Mobile SoCs:

- Snapdragon 888 and newer
- MediaTek Dimensity 9000 and newer
- ARM Mali-G615 and newer

As of January 2023, Apple and Raspberry Pi GPUs do not support variable rate shading.

Using variable rate shading in Godot

Note

Both Forward Plus and Forward Mobile rendering methods support variable rate shading. VRS can be used in both pancake (non-XR) and XR display modes.

The Compatibility rendering method does **not** support variable rate shading.

In the advanced Project Settings, the **Rendering > VRS** section offers settings to control variable rate shading on the root viewport:

- **Mode:** Controls the variable rate shading mode. **Disabled** disables variable rate shading. **Texture** uses a manually

authored texture to set shading density (see the property below). **XR** automatically generates a texture suited for foveated rendering in virtual/augmented reality.
- **Texture:** The texture to use to control shading density on the root viewport. Only used if **Mode** is **Texture**.

For custom viewports, the VRS mode and texture must be set manually to the Viewport node.

Note

On unsupported hardware, there is no visual difference when variable rate shading is enabled. You can check whether hardware supports variable rate shading by running the editor or project with the `--verbose` command line argument.

Creating a VRS density map

If using the **Texture** VRS mode, you *must* set a texture to be used as a density map. Otherwise, no effect will be visible.

You can create your own VRS density map manually using an image editor, or generate it using another method (e.g. on the CPU using the Image class, or on the GPU using a shader). However, beware of performance implications when generating a VRS image dynamically. If opting for dynamic generation, make sure the VRS image generation process is fast enough to avoid outweighing the performance gains from VRS.

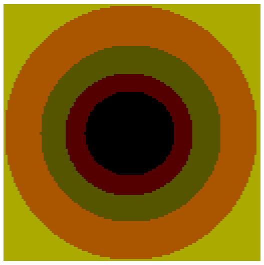
The texture must follow these rules:

- The texture *must* use a lossless compression format so that colors can be matched precisely.
- The following VRS densities are mapped to various colors, with brighter colors representing a lower level of shading precision:

Density	**Color**	**Comment**
1×1 (highest detail)	`rgb(0, 0, 0) - #000000`	
1×2	`rgb(0, 85, 0) - #005500`	
2×1	`rgb(85, 0, 0) - #550000`	
2×2	`rgb(85, 85, 0) - #555500`	
2×4	`rgb(85, 170, 0) - #55aa00`	
4×2	`rgb(170, 85, 0) - #aa5500`	
4×4	`rgb(170, 170, 0) - #aaaa00`	
4×8	`rgb(170, 255, 0) - #aaff00`	Not supported on most hardware.

Density	Color	Comment
8×4	`rgb(255, 170, 0)` - `#ffaa00`	Not supported on most hardware.
8×8 (lowest detail)	`rgb(255, 255, 0)` - `#ffff00`	Not supported on most hardware.

For example, this VRS density texture provides the highest shading density in the center of the viewport, and the lowest shading density in the corners:



Example VRS density map texture, simulating foveated rendering

There are no size or aspect ratio requirements for the VRS density texture. However, there is no benefit to using a VRS density map that is larger than the viewport resolution divided by the GPU's *tile size*. The tile size is what determines the smallest area of pixels where the shading density can be changed separately from other tiles. On most GPUs, this tile size is 8×8 pixels. You can view the tile size by running Godot with the `--verbose` command line argument, as it's printed in the VRS debugging information.

Therefore, sticking to a relatively low resolution such as 256×256 (square) or 480×270 (16:9) is recommended. Depending on your use cases, a square texture may be more suited compared to a texture that matches the most common viewport aspect ratio in your project (such as 16:9).

Tip

When using variable rate shading, you can use a negative texture mipmap LOD bias to reduce blurriness in areas with reduced shading rate.

Note that the texture LOD bias is set globally, so this will also affect areas of the viewport with full shading rate. Don't use values that are too low, or textures will appear grainy.

Performance comparison

To give an idea of how much VRS can improve performance in theory, here's a performance comparison with the textured example scene shown at the top of this page. The VRS density map example present on this page is used.

Results were captured on a GeForce RTX 4090 with the NVIDIA 525.60.11 driver.

Resolution	VRS disabled	VRS enabled	Performance improvement
1920×1080 (Full HD)	2832 FPS	3136 FPS	+10.7%
2560×1440 (QHD)	2008 FPS	2256 FPS	+12.3%

Resolution	VRS disabled	VRS enabled	Performance improvement
3840×2160 (4K)	1236 FPS	1436 FPS	+16.2%
7680×4320 (8K)	384 FPS	473 FPS	+23.1%

In terms of performance improvements, variable rate shading is more beneficial at higher target resolutions. The reduction in visual quality is also less noticeable at high resolutions.

Note

For non-VR games, you will probably have to use a less aggressive VRS texture than what was used in this example. As a result, the effective performance gains will be lower.

Prototyping levels with CSG

CSG stands for **Constructive Solid Geometry**, and is a tool to combine basic shapes or custom meshes to create more complex shapes. In 3D modeling software, CSG is mostly known as "Boolean Operators".

Level prototyping is one of the main uses of CSG in Godot. This technique allows users to create the most common shapes by combining primitives. Interior environments can be created by using inverted primitives.

Note

The CSG nodes in Godot are mainly intended for prototyping. There is no built-in support for UV mapping or editing 3D polygons (though extruded 2D polygons can be used with the CSGPolygon3D node). In addition CSG can't reliably create meshes made up of multiple nodes without holes.

If you're looking for an easy to use level design tool for a project, you may want to use FuncGodot [https://github.com/func-godot/func_godot_plugin] or Cyclops Level Builder [https://github.com/blackears/cyclopsLevelBuilder] instead.

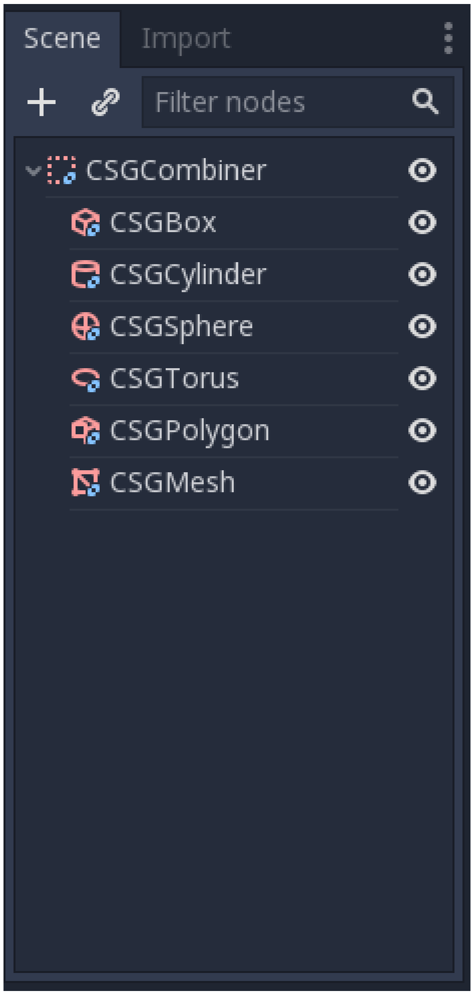
See also

You can check how to use CSG nodes to build various shapes (such as stairs or roads) using the Constructive Solid Geometry demo project [https://github.com/godotengine/godot-demo-projects/tree/master/3d/csg].

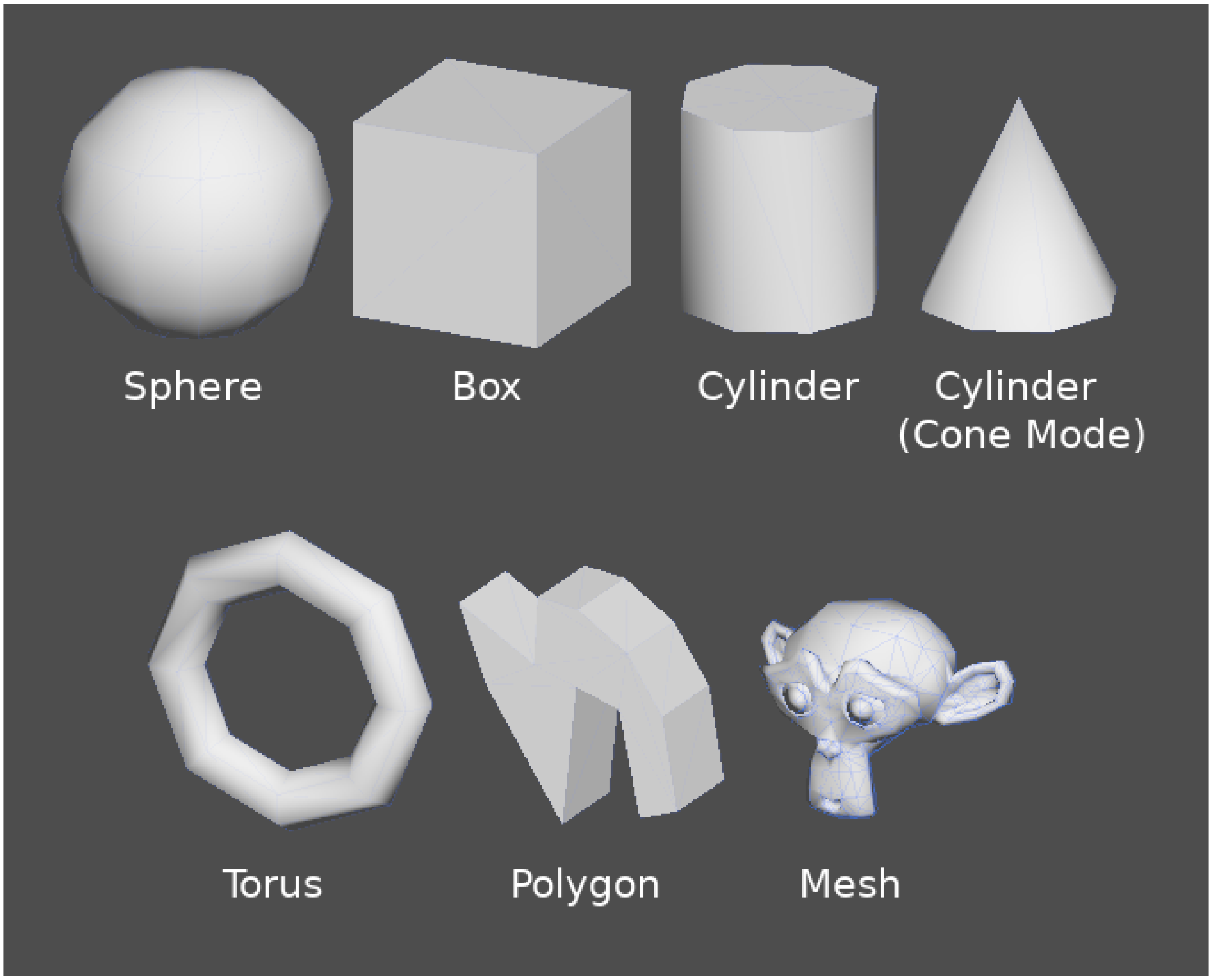
Introduction to CSG nodes

Like other features of Godot, CSG is supported in the form of nodes. These are the CSG nodes:

- CSGBox3D
- CSGCylinder3D (also supports cone)
- CSGSphere3D
- CSGTorus3D
- CSGPolygon3D
- CSGMesh3D
- CSGCombiner3D


Scene
Import
Filter nodes

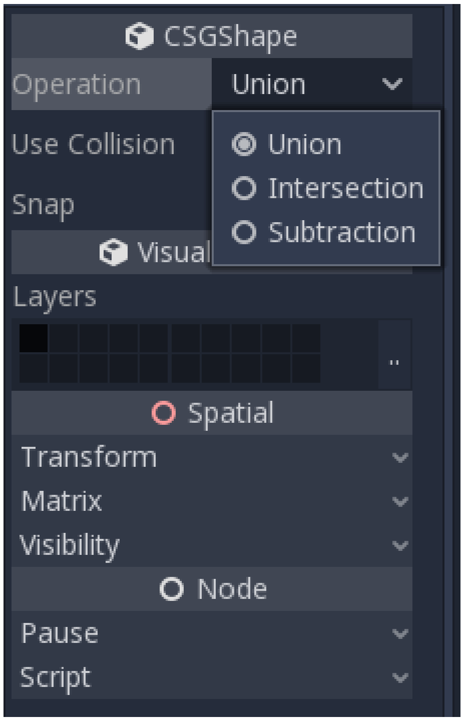
CSGCombiner
- CSGBox
- CSGCylinder
- CSGSphere
- CSGTorus
- CSGPolygon
- CSGMesh


Sphere
Box
Cylinder
Cylinder
(Cone Mode)
Torus
Polygon
Mesh

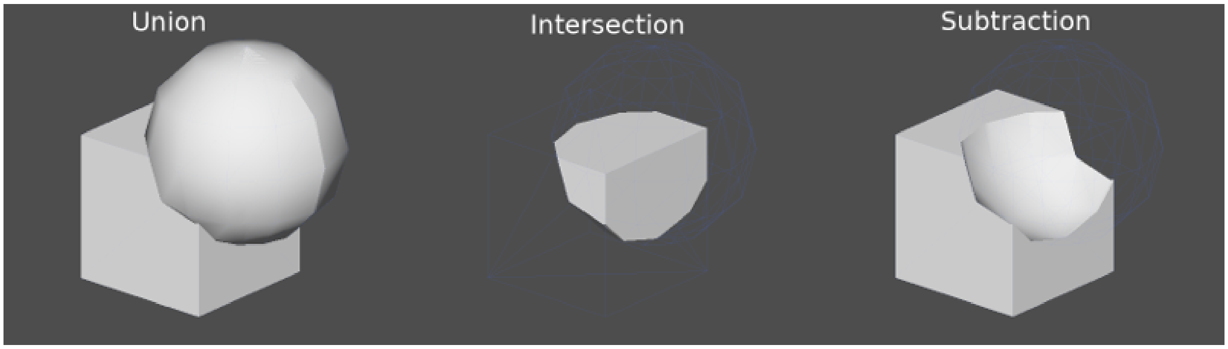
CSG tools features

Every CSG node supports 3 kinds of boolean operations:

- **Union:** Geometry of both primitives is merged, intersecting geometry is removed.
- **Intersection:** Only intersecting geometry remains, the rest is removed.
- **Subtraction:** The second shape is subtracted from the first, leaving a dent with its shape.


CSGShape
Use Collision
Snap
Visual
Union
Intersection
Subtraction
Layers
Spatial
Transform
Matrix
Visibility
Node
Pause
Script

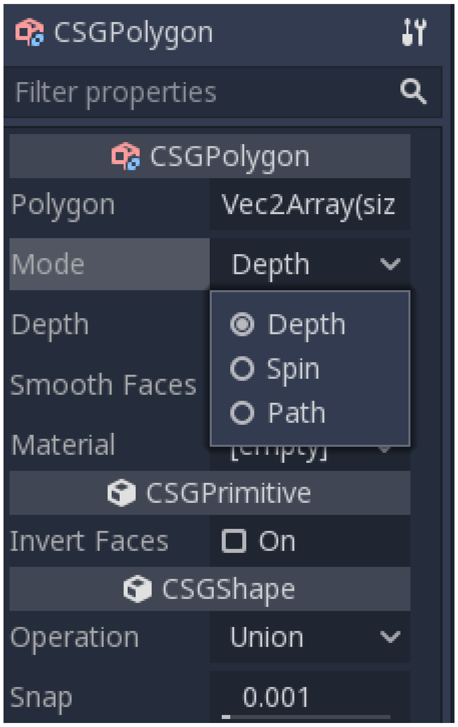
Operation Union


Union
Intersection
Subtraction

CSGPolygon

The CSGPolygon3D node extrude along a Polygon drawn in 2D (in X, Y coordinates) in the following ways:

- **Depth:** Extruded back a given amount.
- **Spin:** Extruded while spinning around its origin.
- **Path:** Extruded along a Path node. This operation is commonly called lofting.



CSGPolygon

Filter properties

CSGPolygon

Polygon Vec2Array(siz

Mode Depth

Depth

Smooth Faces

Material [empty]

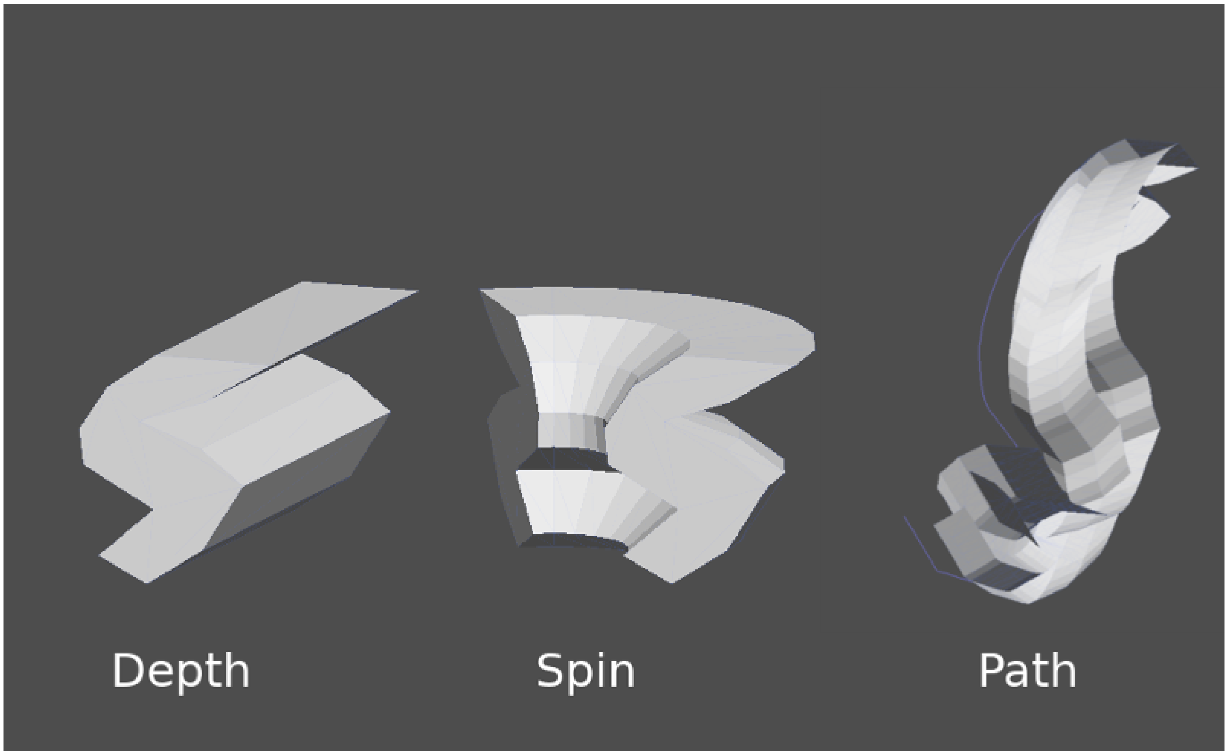
CSGPrimitive

Invert Faces ☐ On

CSGShape

Operation Union

Snap 0.001



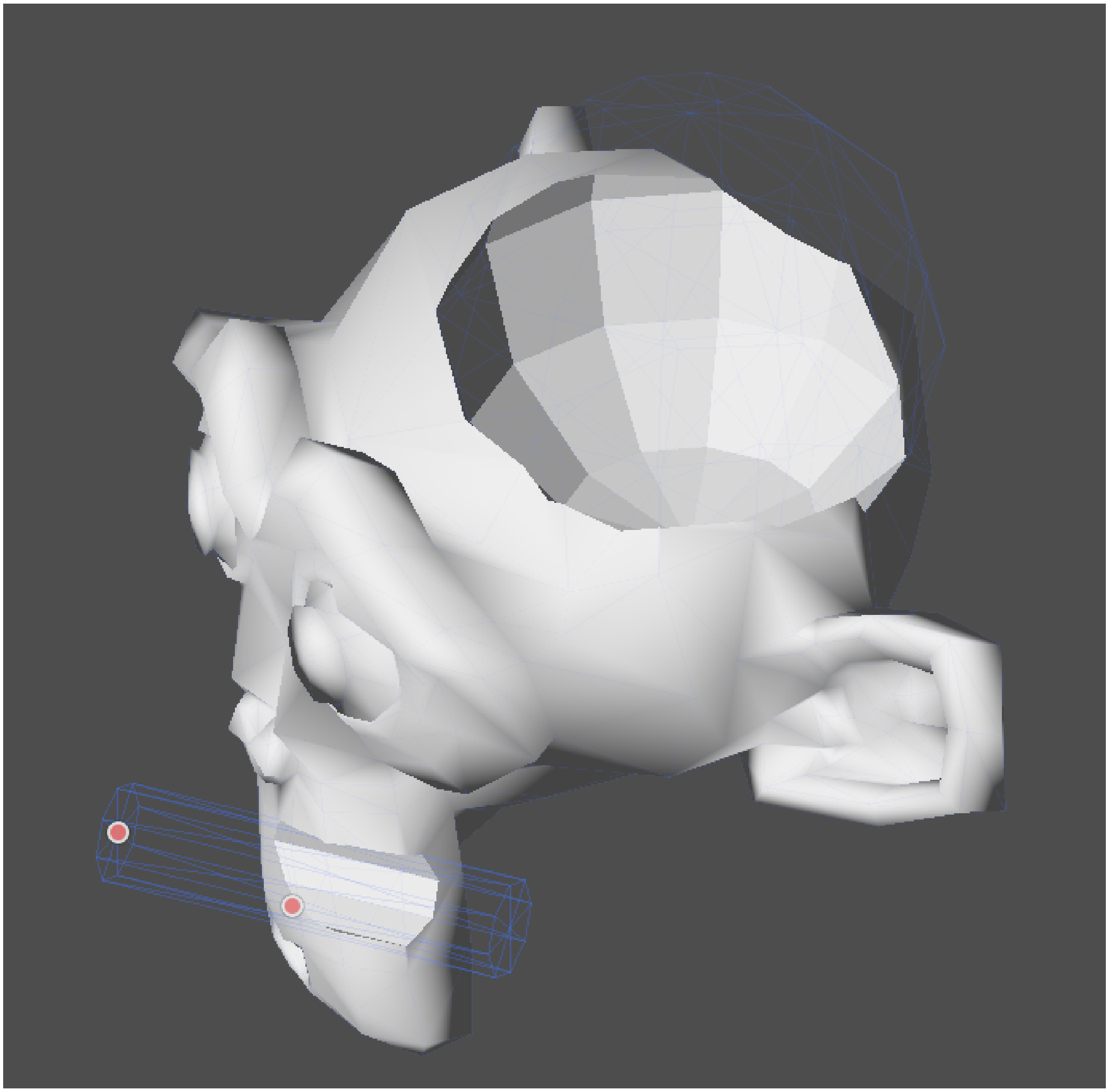
Note

The **Path** mode must be provided with a Path3D node to work. In the Path node, draw the path and the polygon in CSGPolygon3D will extrude along the given path.

Custom meshes

Any mesh can be used for CSGMesh3D; the mesh can be modeled in other software and imported into Godot. Multiple materials are supported. There are some restrictions for geometry:

- it must be closed,
- it must not self-intersect,
- it must not contain internal faces,
- every edge must connect to only two other faces.



CSGCombiner3D

The CSGCombiner3D node is an empty shape used for organization. It will only combine children nodes.

Processing order

Every CSG node will first process its children nodes and their operations: union, intersection, or subtraction, in tree order, and apply them to itself one after the other.

Note

In the interest of performance, make sure CSG geometry remains relatively simple, as complex meshes can take a while to process. If adding objects together (such as table and room objects), create them as separate CSG trees. Forcing too many objects in a single tree will eventually start affecting performance. Only use binary operations where you actually need them.

Prototyping a level

We will prototype a room to practice the use of CSG tools.

Tip

Working in **Orthogonal** projection gives a better view when combining the CSG shapes.

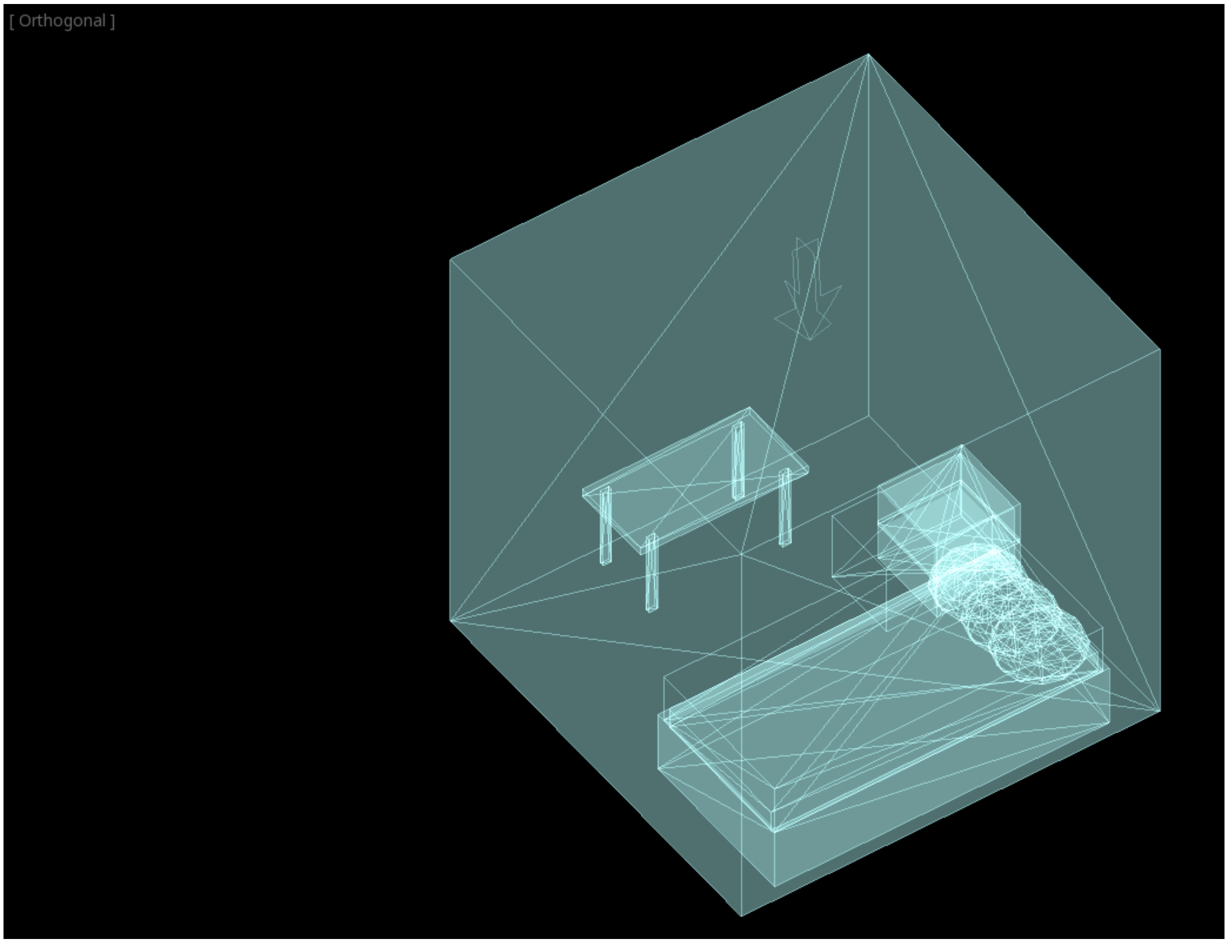
Our level will contain these objects:

- a room,
- a bed,
- a lamp,
- a desk,
- a bookshelf.

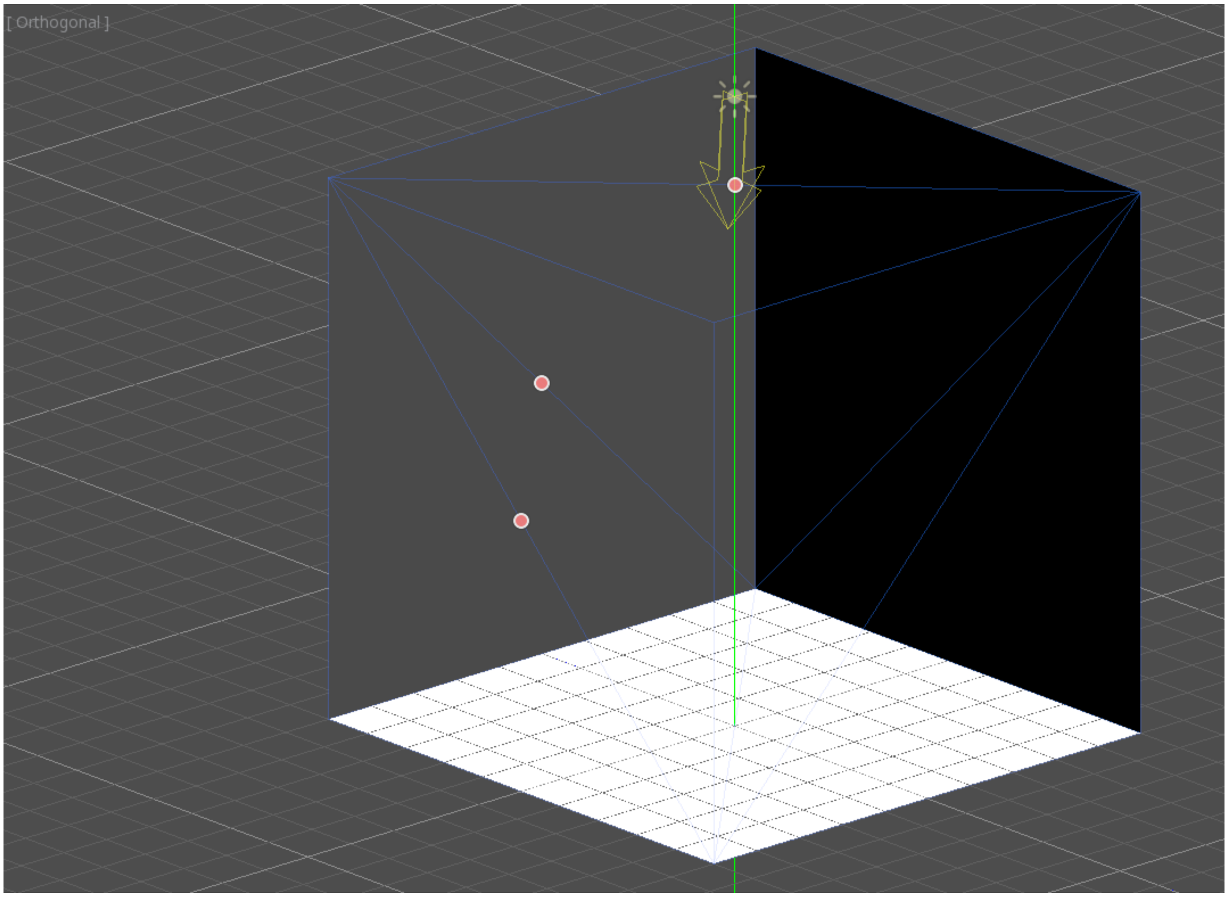
Create a scene with a Node3D node as root node.

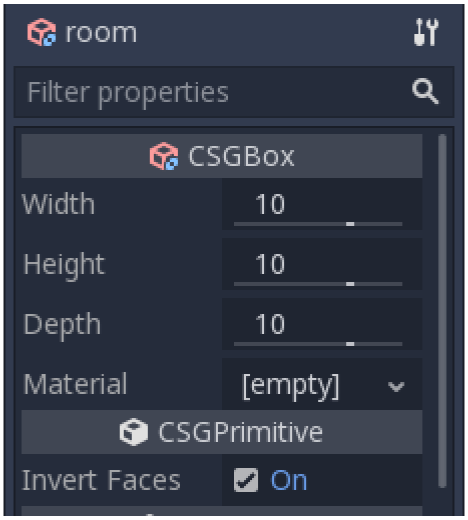
Tip

The default lighting of the environment doesn't provide clear shading at some angles. Change the display mode using **Display Overdraw** in the 3D viewport menu, or add a DirectionalLight node to help you see clearly.



Create a CSGBox3D and name it `room`, enable **Invert Faces** and change the dimensions of your room.


[Orthogonal]


room
Filter properties

CSGBox

Width 10

Height 10

Depth 10

Material [empty]

CSGPrimitive

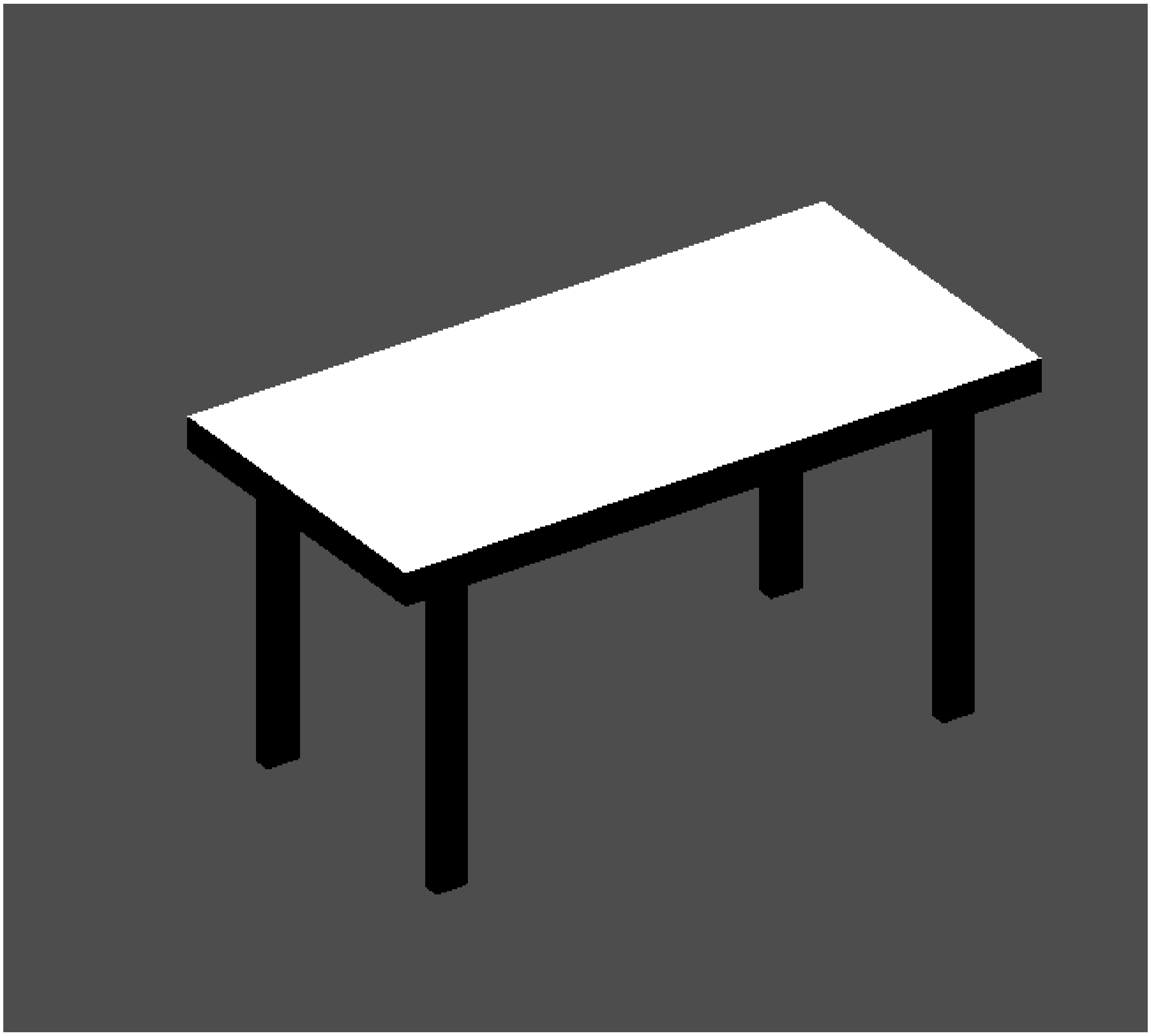
Invert Faces ☑ On

Next, create a CSGCombiner3D and name it `desk`.

A desk has one surface and 4 legs:

- Create 1 CSGBox3D children node in **Union** mode for the surface and adjust the dimensions.
- Create 4 CSGBox3D children nodes in **Union** mode for the legs and adjust the dimensions.

Adjust their placement to resemble a desk.

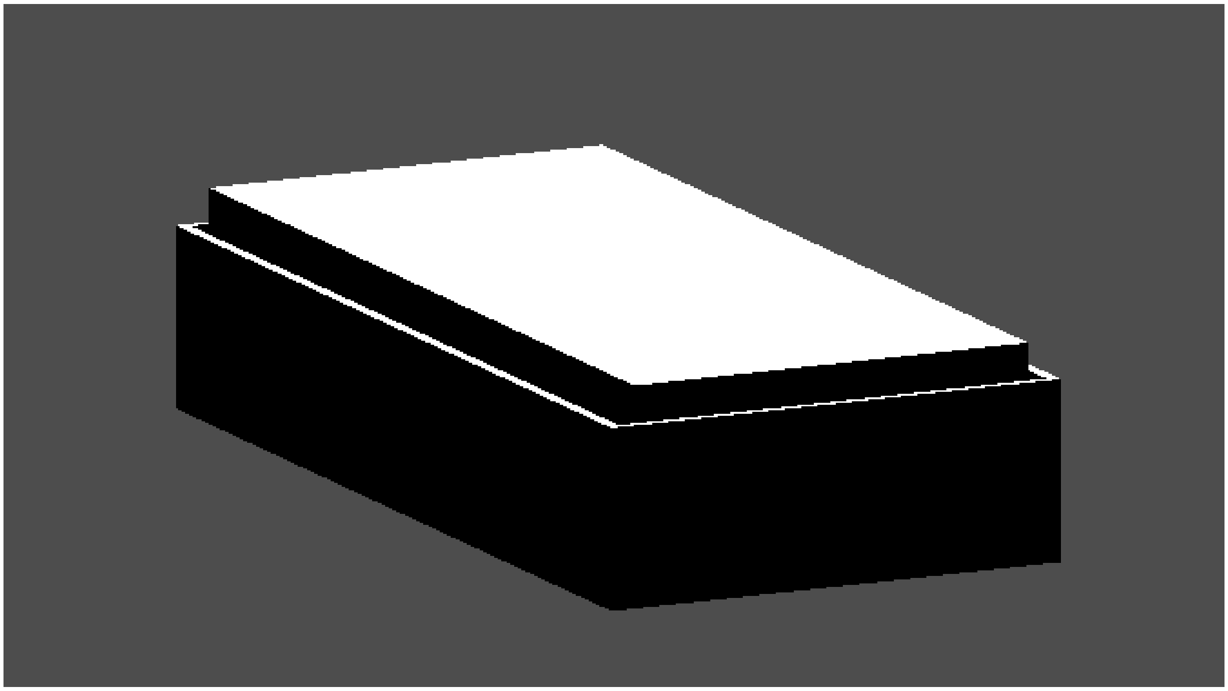


Note

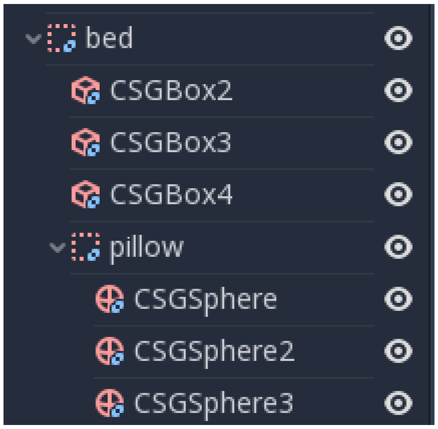
CSG nodes inside a CSGCombiner3D will only process their operation within the combiner. Therefore, CSGCombiner3Ds are used to organize CSG nodes.

Create a CSGCombiner3D and name it `bed`.

Our bed consists of 3 parts: the bed, the mattress and a pillow. Create a CSGBox3D and adjust its dimension for the bed. Create another CSGBox3D and adjust its dimension for the mattress.

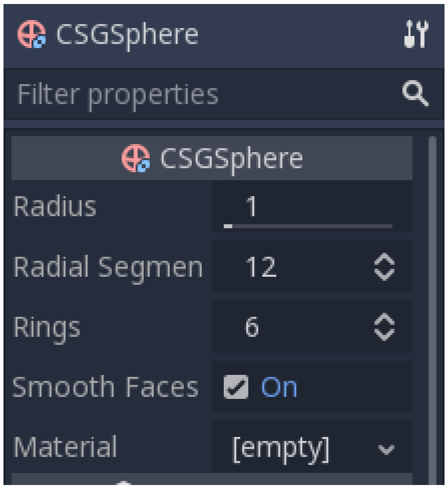


We will create another CSGCombiner3D named `pillow` as the child of `bed`. The scene tree should look like this:



- bed
  - CSGBox2
  - CSGBox3
  - CSGBox4
  - pillow
    - CSGSphere
    - CSGSphere2
    - CSGSphere3

We will combine 3 CSGSphere3D nodes in **Union** mode to form a pillow. Scale the Y axis of the spheres and enable **Smooth Faces**.



CSGSphere

CSGSphere

Radius	1
Radial Segmen	12
Rings	6
Smooth Faces	☑ On
Material	[empty]

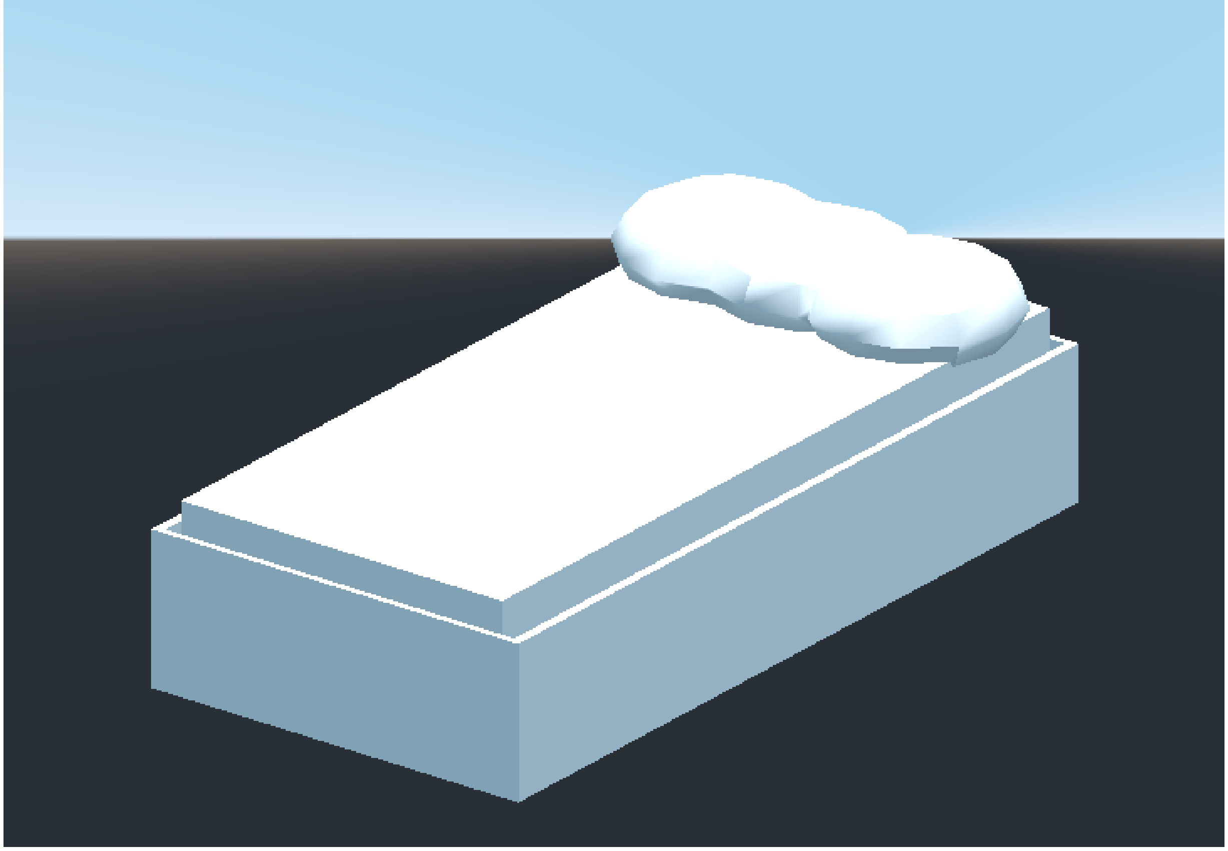
Select the `pillow` node and switch the mode to **Subtraction**; the combined spheres will cut a hole into the mattress.

Try to re-parent the `pillow` node to the root `Node3D` node; the hole will disappear.

Note

This is to illustrate the effect of CSG processing order. Since the root node is not a CSG node, the CSGCombiner3D nodes are the end of the operations; this shows the use of CSGCombiner3D to organize the CSG scene.

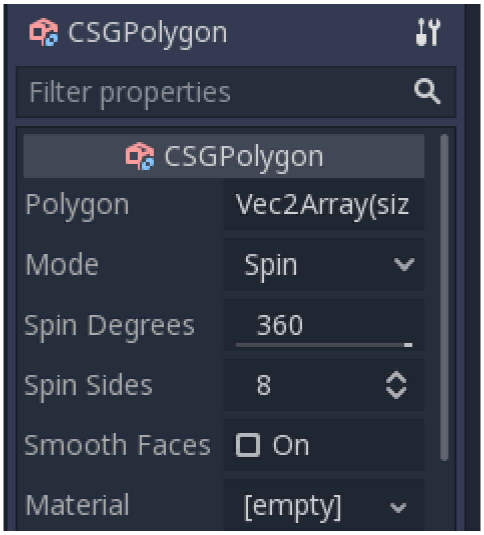
Undo the re-parent after observing the effect. The bed you've built should look like this:



Create a CSGCombiner3D and name it `lamp`.

A lamp consists of 3 parts: the stand, the pole and the lampshade. Create a CSGCylinder3D, enable the **Cone** option and make it the stand. Create another CSGCylinder3D and adjust the dimensions to use it as a pole.

We will use a CSGPolygon3D for the lampshade. Use the **Spin** mode for the CSGPolygon3D and draw a [trapezoid](https://en.wikipedia.org/wiki/Trapezoid) [https://en.wikipedia.org/wiki/Trapezoid] while in **Front View** (numeric keypad 1); this shape will extrude around the origin and form the lampshade.



CSGPolygon

Filter properties

CSGPolygon

Polygon Vec2Array(siz

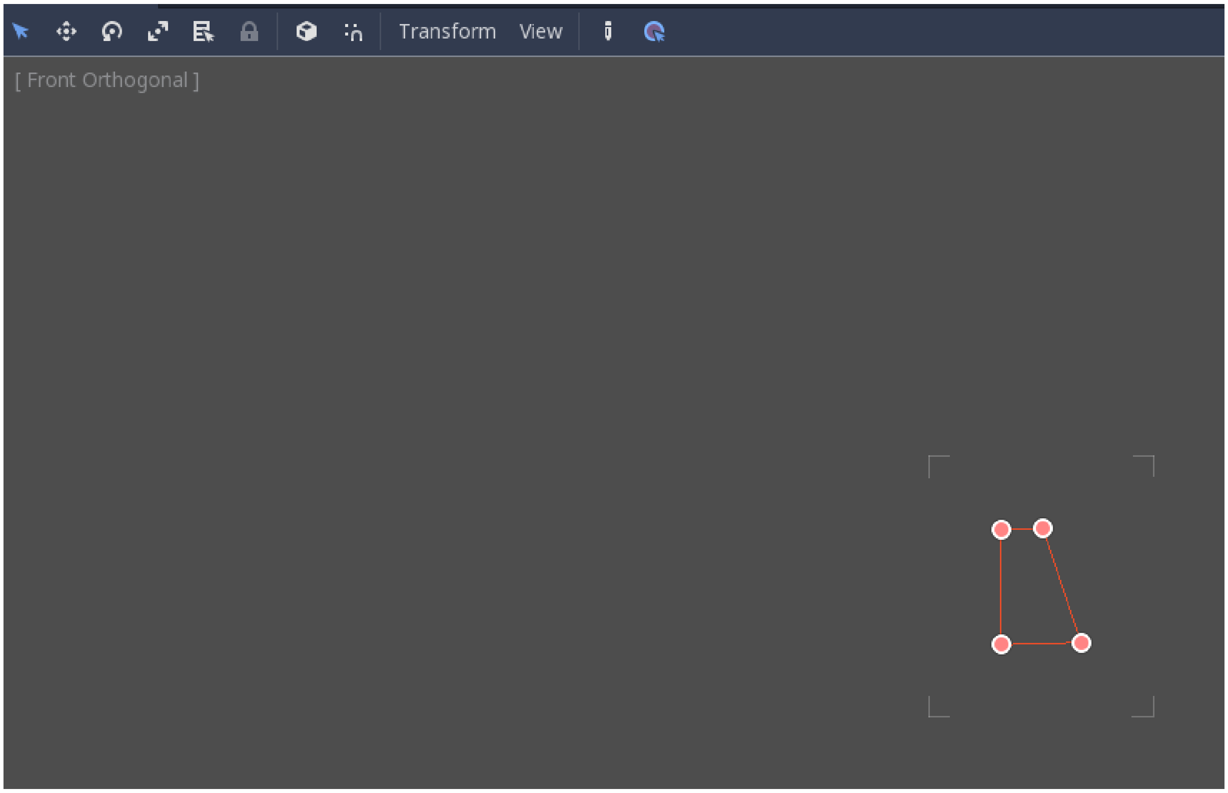
Mode Spin

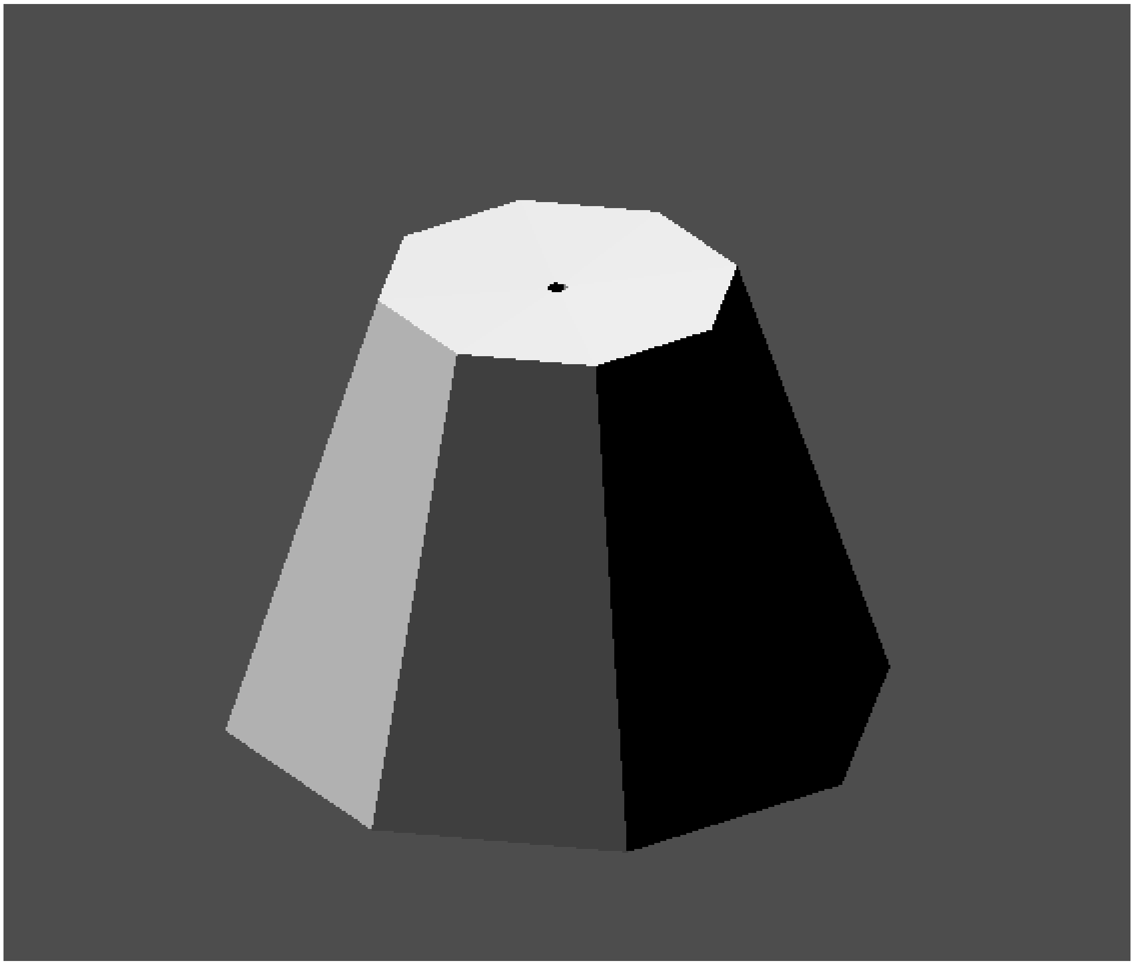
Spin Degrees 360

Spin Sides 8

Smooth Faces ☐ On

Material [empty]

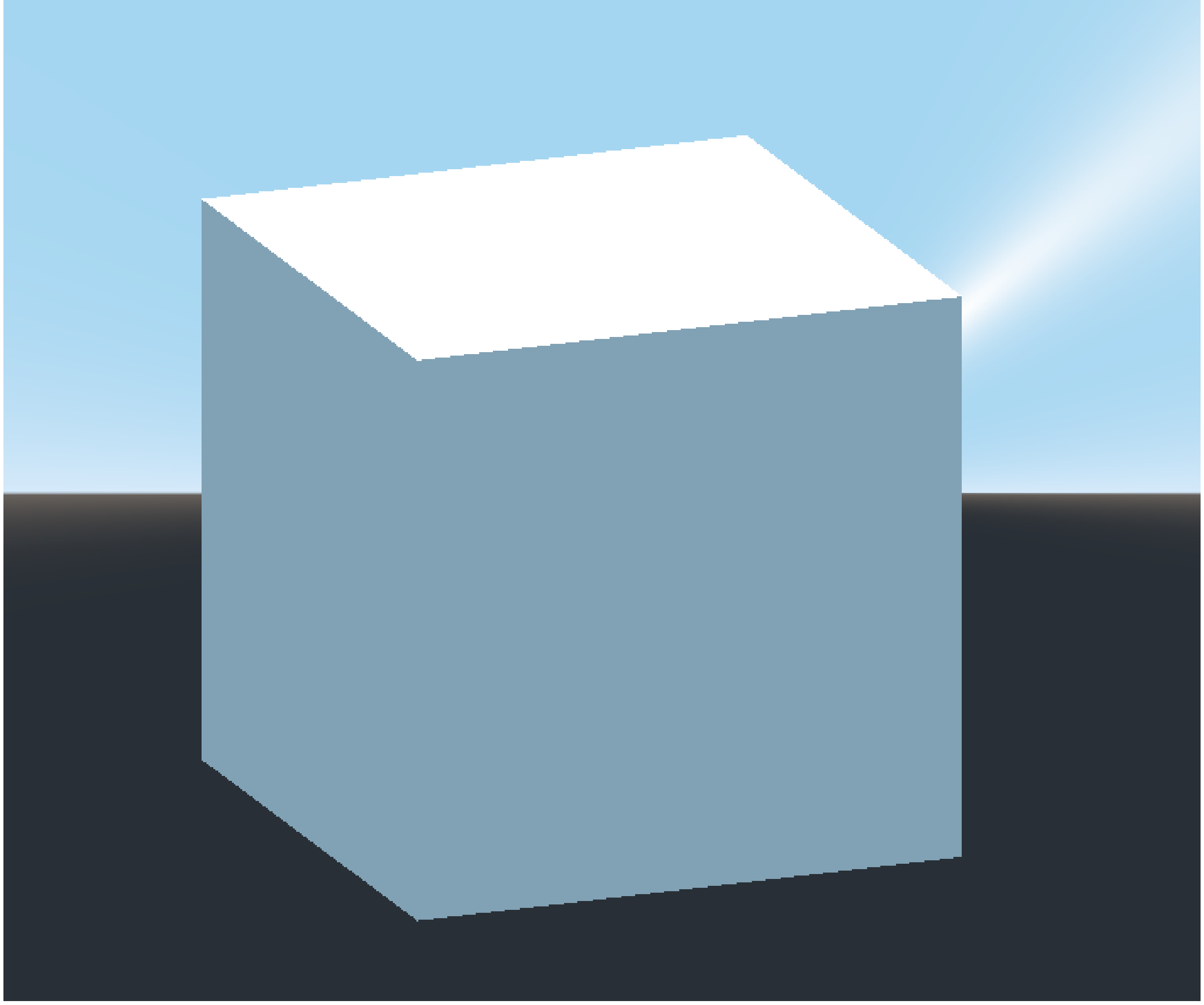

Transform
View
[Front Orthogonal]



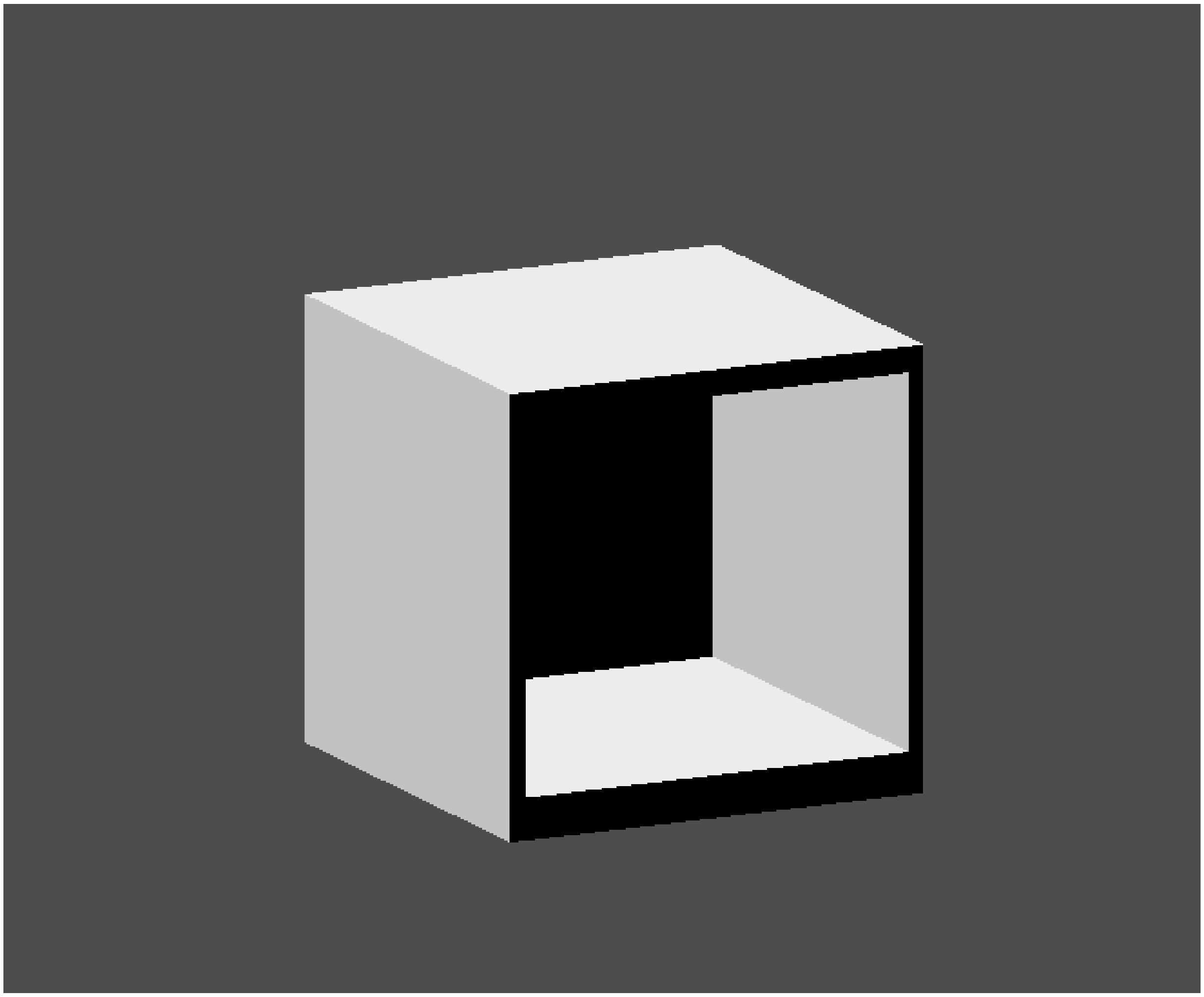
Adjust the placement of the 3 parts to make it look like a lamp.

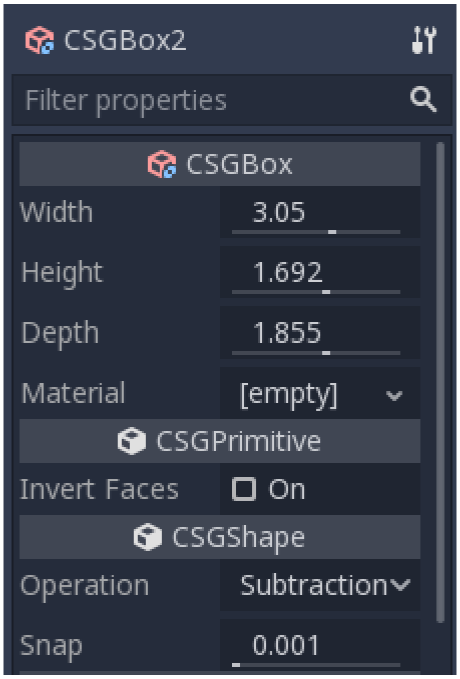
Create a CSGCombiner3D and name it `bookshelf`.

We will use 3 CSGBox3D nodes for the bookshelf. Create a CSGBox3D and adjust its dimensions; this will be the size of the bookshelf.



Duplicate the CSGBox3D and shorten the dimensions of each axis and change the mode to **Subtraction**.





CSGBox2
Filter properties

CSGBox

Width 3.05

Height 1.692

Depth 1.855

Material [empty]

CSGPrimitive

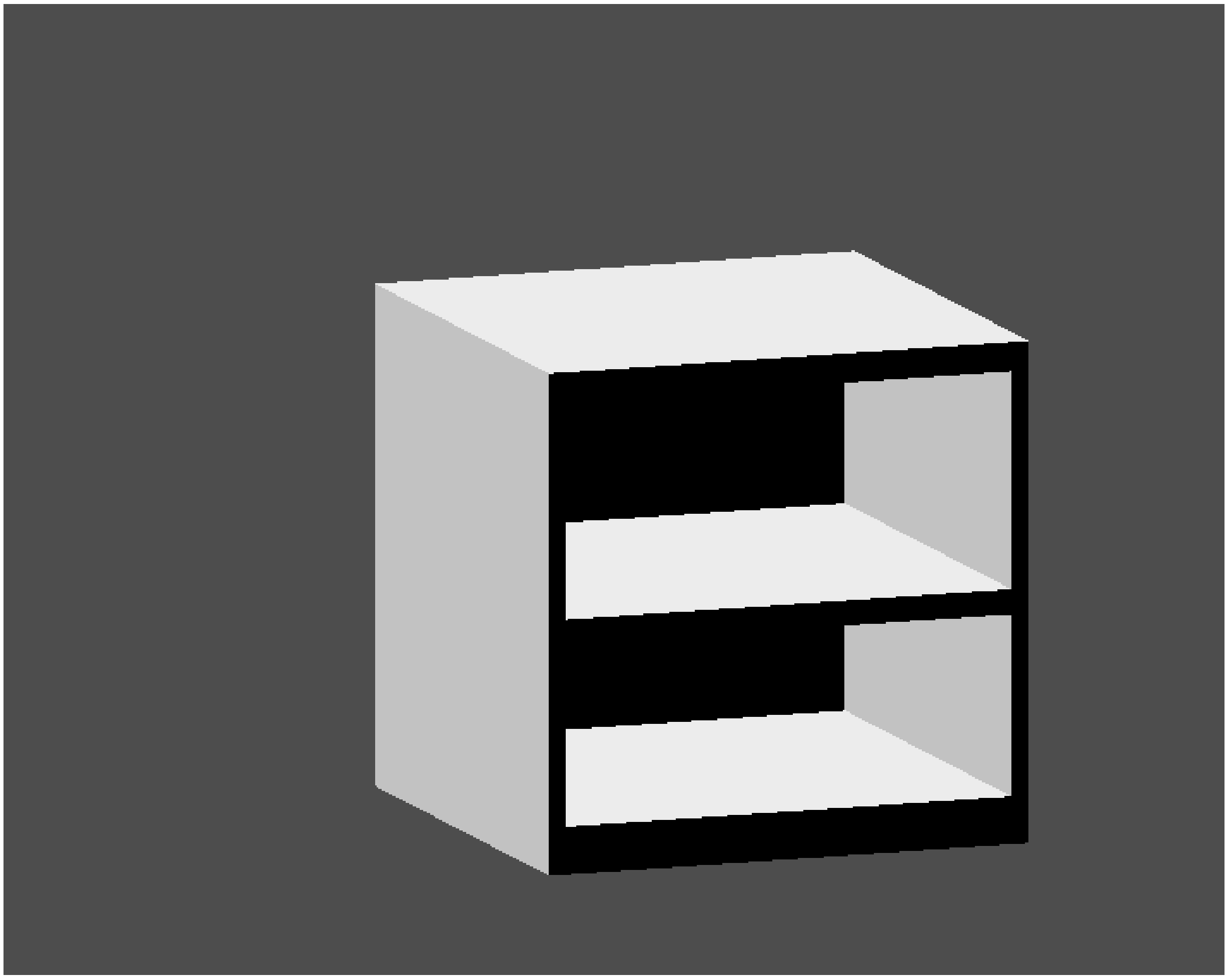
Invert Faces ☐ On

CSGShape

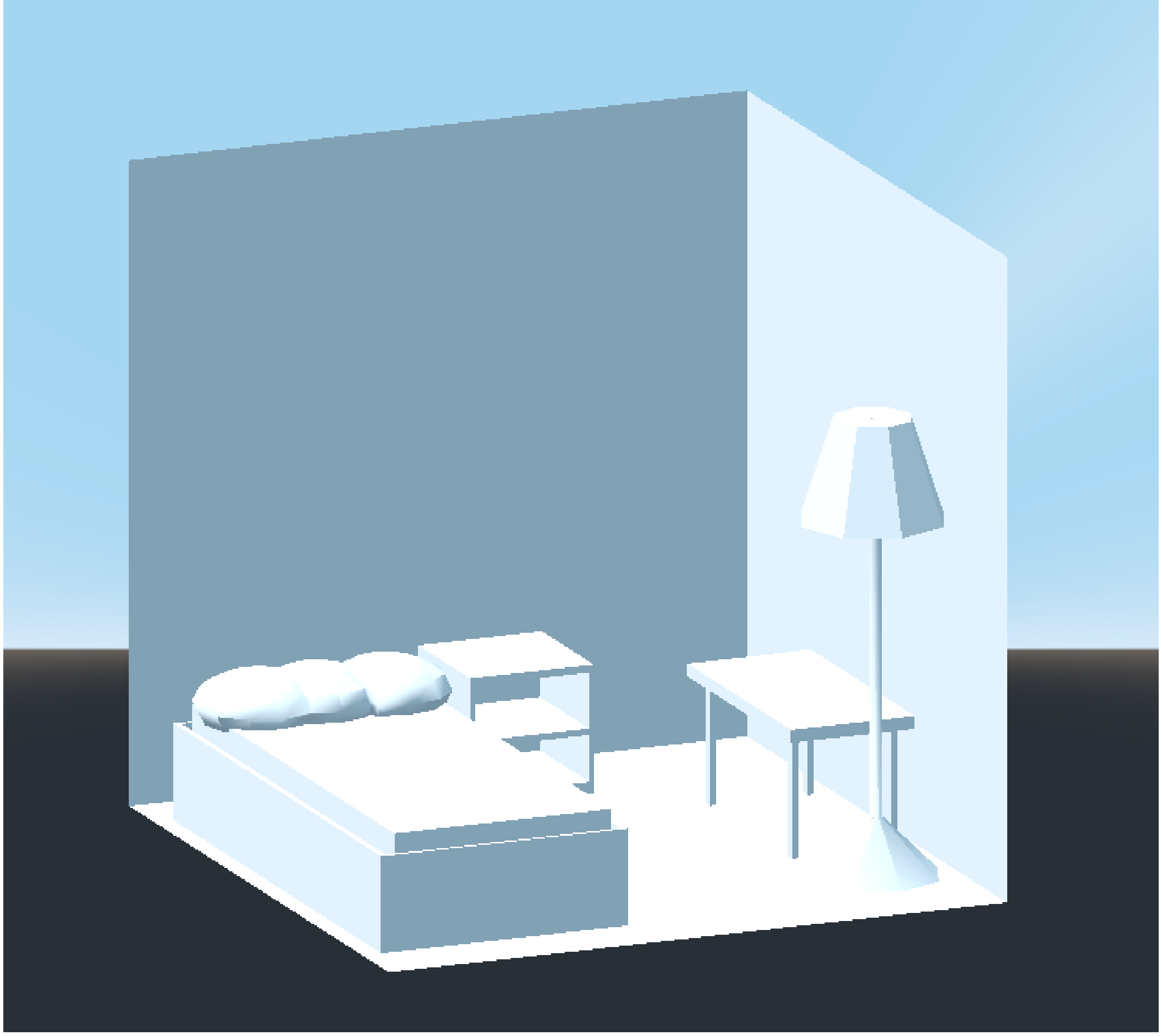
Operation Subtraction

Snap 0.001

You've almost built a shelf. Create one more CSGBox3D for dividing the shelf into two levels.



Position your furniture in your room as you like and your scene should look this:



You've successfully prototyped a room level with the CSG tools in Godot. CSG tools can be used for designing all kinds of levels, such as a maze or a city; explore its limitations when designing your game.

Using prototype textures

Godot's Standard Material 3D and ORM Material 3D supports *triplanar mapping*, which can be used to automatically apply a texture to arbitrary objects without distortion. This is handy when using CSG as Godot doesn't support editing UV maps on CSG nodes yet. Triplanar mapping is relatively slow, which usually restricts its usage to organic surfaces like terrain. Still, when prototyping, it can be used to quickly apply textures to CSG-based levels.

Note

If you need some textures for prototyping, Kenney made a set of CC0-licensed prototype textures [https://kenney.nl/assets/prototype-textures].

There are two ways to apply a material to a CSG node:

- Applying it to a CSGCombiner3D node as a material override (**Geometry > Material Override** in the Inspector). This will affect its children automatically, but will make it impossible to change the material in individual children.
- Applying a material to individual nodes (**Material** in the Inspector). This way, each CSG node can have its own appearance. Subtractive CSG nodes will apply their material to the nodes they're "digging" into.

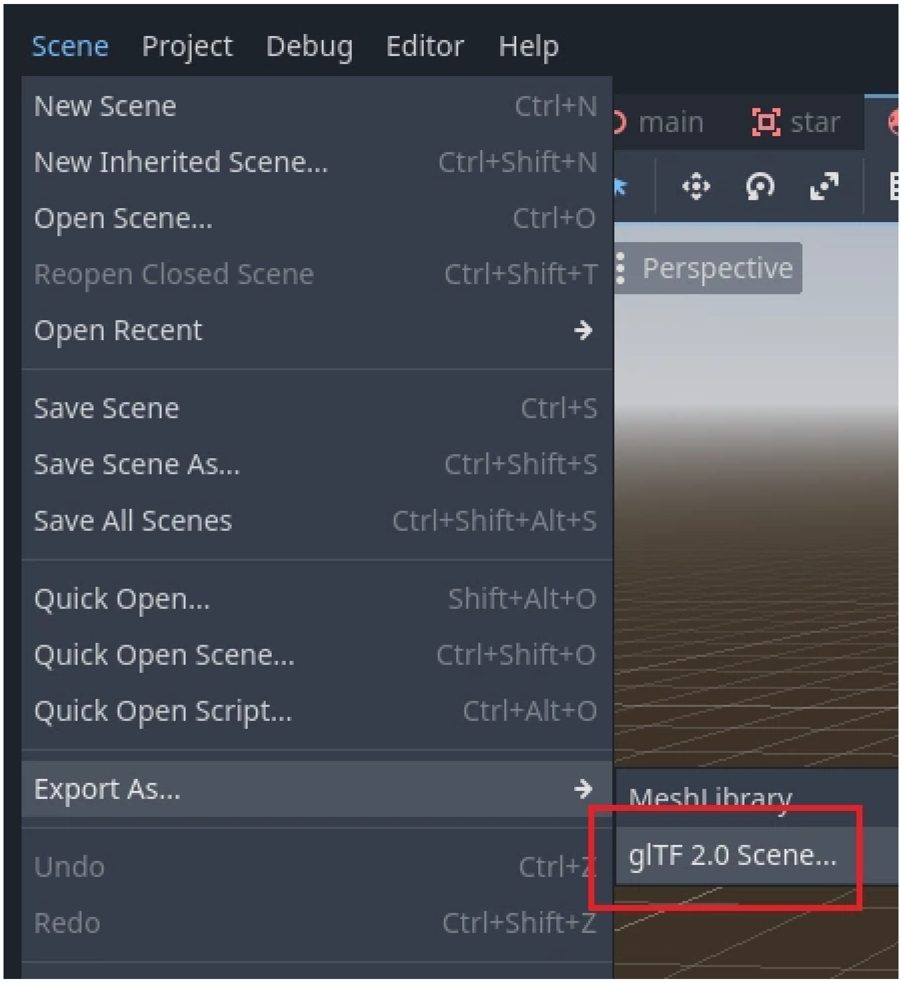
To apply triplanar mapping to a CSG node, select it, go to the Inspector, click the **[empty]** text next to **Material Override** (or **Material** for individual CSG nodes). Choose **New StandardMaterial3D**. Click the newly created material's icon to edit it. Unfold the **Albedo** section and load a texture into the **Texture** property. Now, unfold the **Uv1** section and check **Triplanar**. You can change the texture offset and scale on each axis by playing with the **Scale** and **Offset** properties just above. Higher values in the **Scale** property will cause the texture to repeat more often.

Tip

You can copy a StandardMaterial3D to reuse it across CSG nodes. To do so, click the dropdown arrow next to a material property in the Inspector and choose **Copy**. To paste it, select the node you'd like to apply the material onto, click the dropdown arrow next to its material property then choose **Paste**.

Exporting as glTF

It can be useful to block out a level using CSG, then export it as a 3d model, to import into 3D modeling software. You can do this by selecting **Scene > Export As... > glTF 2.0 Scene**.


Scene
Project
Debug
Editor
Help
New Scene Ctrl+N
New Inherited Scene... Ctrl+Shift+N
Open Scene... Ctrl+O
Reopen Closed Scene Ctrl+Shift+T
Open Recent
Save Scene Ctrl+S
Save Scene As... Ctrl+Shift+S
Save All Scenes Ctrl+Shift+Alt+S
Quick Open... Shift+Alt+O
Quick Open Scene... Ctrl+Shift+O
Quick Open Script... Ctrl+Alt+O
Export As...
Undo Ctrl+Z
Redo Ctrl+Shift+Z
main
star
Perspective
MeshLibrary
glTF 2.0 Scene...

Using GridMaps

Introduction

Gridmaps are a tool for creating 3D game levels, similar to the way TileMap works in 2D. You start with a predefined collection of 3D meshes (a MeshLibrary) that can be placed on a grid, as if you were building a level with an unlimited amount of Lego blocks.

Collisions and navigation can also be added to the meshes, just like you would do with the tiles of a tilemap.

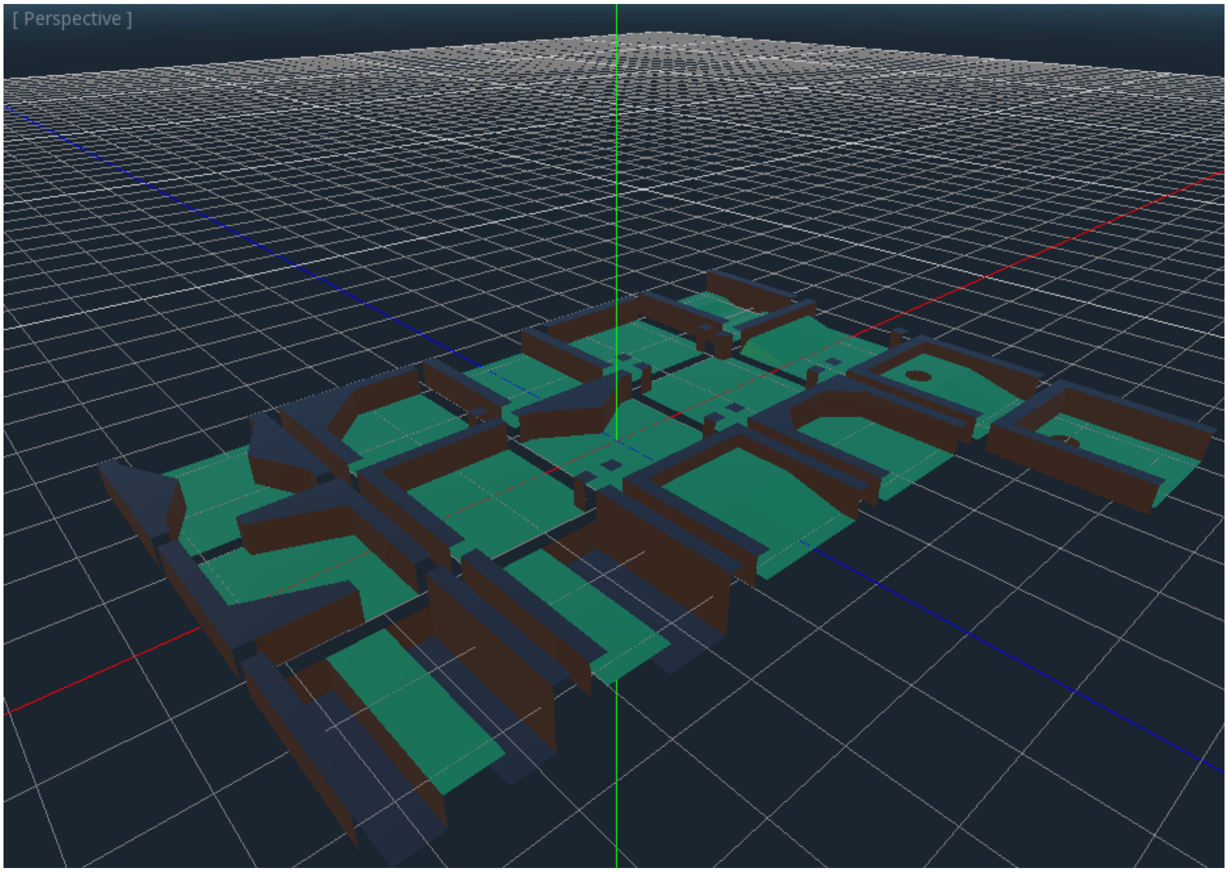
Example project

To learn how GridMaps work, start by downloading the sample project: gridmap_starter.zip [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/gridmap_starter.zip].

Unzip this project and add it to the Project Manager using the "Import" button.

Creating a MeshLibrary

To begin, you need a MeshLibrary, which is a collection of individual meshes that can be used in the gridmap. Open the "mesh_library_source.tscn" scene to see an example of how to set up the mesh library.

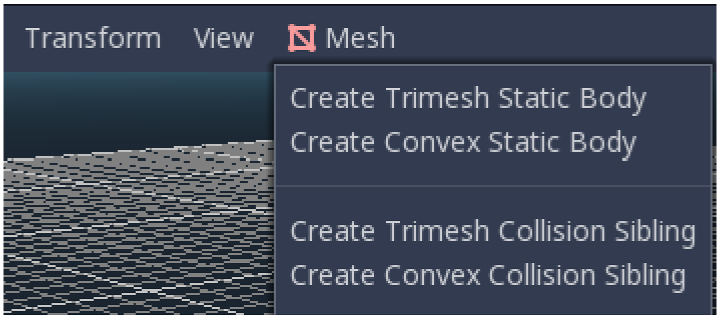


As you can see, this scene has a Node3D node as its root, and a number of MeshInstance3D node children.

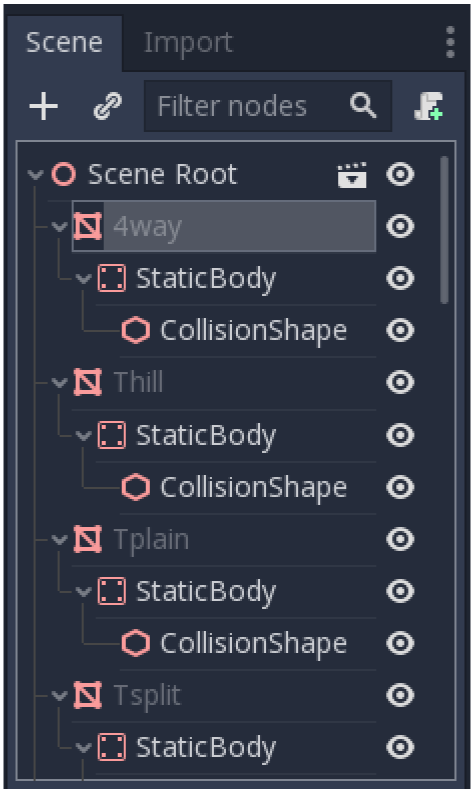
If you don't need any physics in your scene, then you're done. However, in most cases you'll want to assign collision bodies to the meshes.

Collisions

You can manually assign a StaticBody3D and CollisionShape3D to each mesh. Alternatively, you can use the "Mesh" menu to automatically create the collision body based on the mesh data.


Transform
View
Mesh
Create Trimesh Static Body
Create Convex Static Body
Create Trimesh Collision Sibling
Create Convex Collision Sibling

Note that a "Convex" collision body will work better for simple meshes. For more complex shapes, select "Create Trimesh Static Body". Once each mesh has a physics body and collision shape assigned, your mesh library is ready to be used.


Scene
Import
Filter nodes
Scene Root
4way
StaticBody
CollisionShape
Thill
StaticBody
CollisionShape
Tplain
StaticBody
CollisionShape
Tsplit
StaticBody

Materials

Only the materials from within the meshes are used when generating the mesh library. Materials set on the node will be ignored.

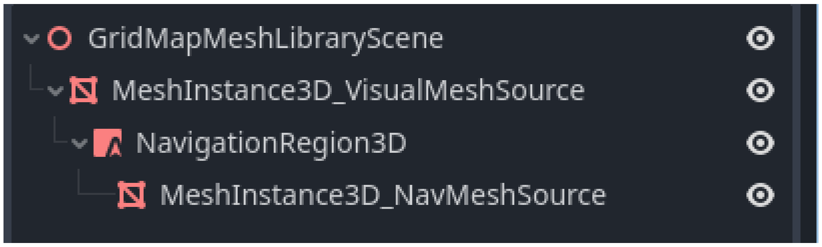
NavigationMeshes

Like all mesh instances, MeshLibrary items can be assigned a NavigationMesh resource, which can be created manually, or baked as described below.

To create the NavigationMesh from a MeshLibrary scene export, place a NavigationRegion3D child node below the main MeshInstance3D for the GridMap item. Add a valid NavigationMesh resource to the NavigationRegion3D and some source geometry nodes below and bake the NavigationMesh.

Note

With small grid cells it is often necessary to reduce the NavigationMesh properties for agent radius and region minimum size.


GridMapMeshLibraryScene
MeshInstance3D_VisualMeshSource
NavigationRegion3D
MeshInstance3D_NavMeshSource

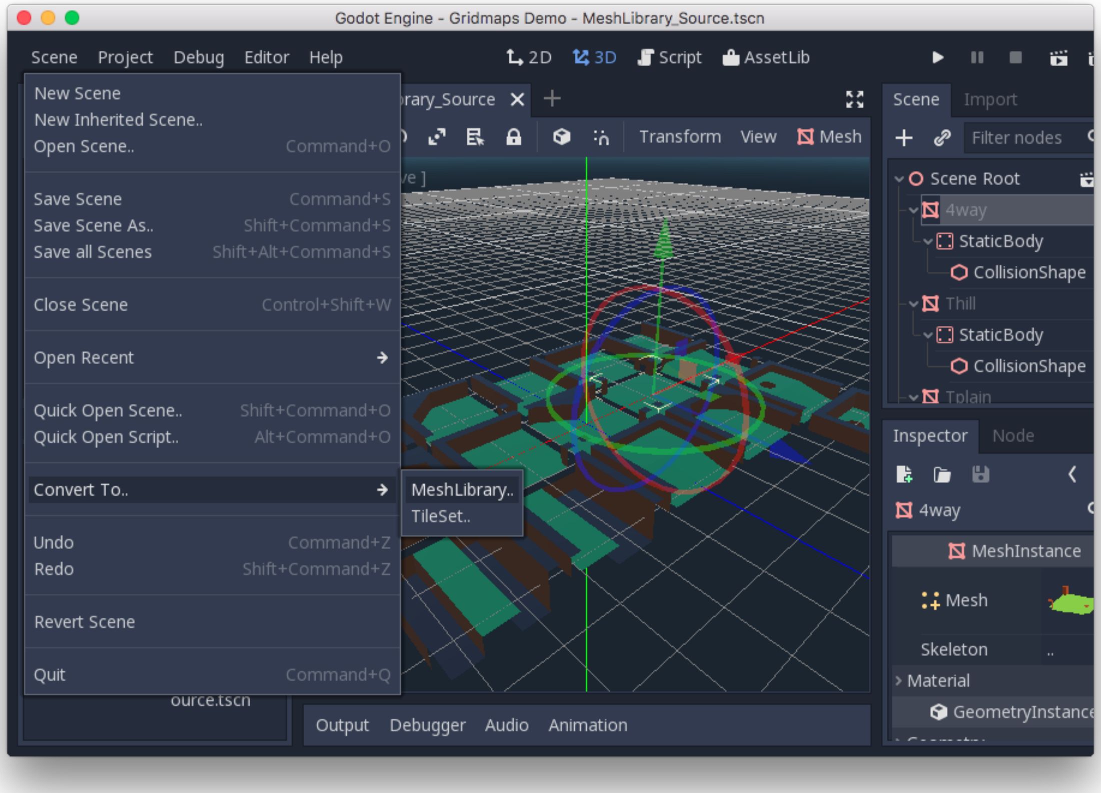
Nodes below the NavigationRegion3D are ignored for the MeshLibrary scene export, so additional nodes can be added as source geometry just for baking the navmesh.

Warning

The baked cell size of the NavigationMesh must match the NavigationServer map cell size to properly merge the navigation meshes of different grid cells.

Exporting the MeshLibrary

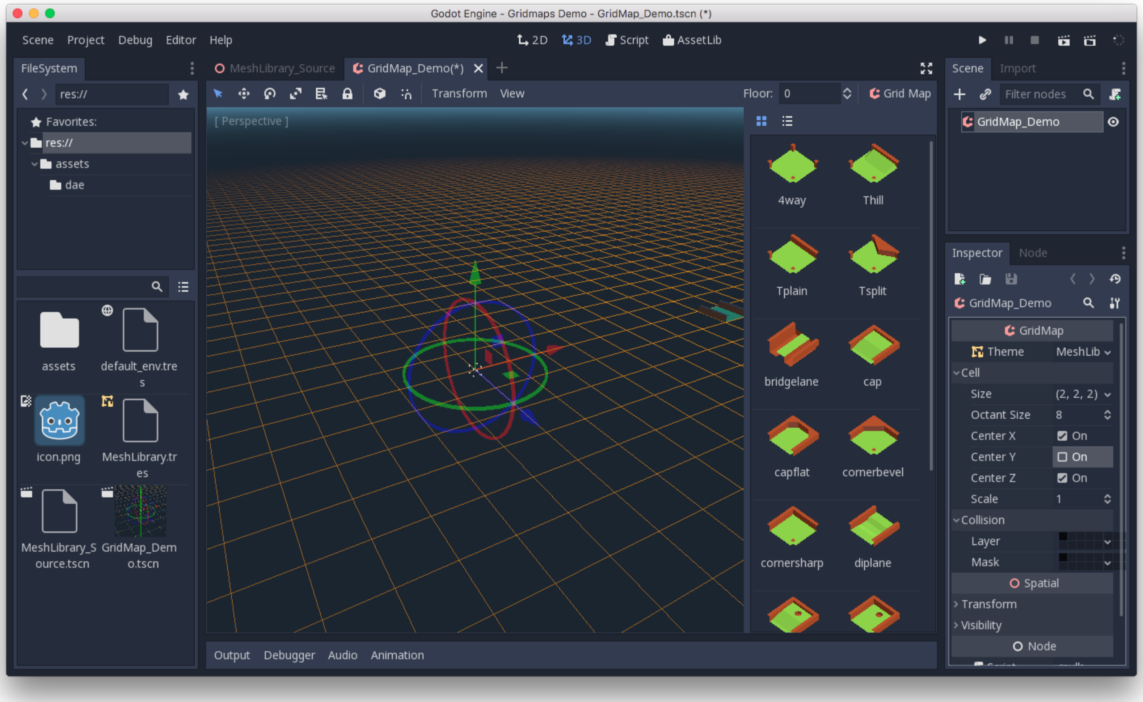
To export the library, click on **Scene > Export As... > MeshLibrary...**, and save it as a resource.


Godot Engine - Gridmaps Demo - MeshLibrary_Source.tscn
Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
New Scene
New Inherited Scene..
Open Scene.. Command+O
Save Scene Command+S
Save Scene As.. Shift+Command+S
Save all Scenes Shift+Alt+Command+S
Close Scene Control+Shift+W
Open Recent
Quick Open Scene.. Shift+Command+O
Quick Open Script.. Alt+Command+O
Convert To..
MeshLibrary..
TileSet..
Undo Command+Z
Redo Shift+Command+Z
Revert Scene
Quit Command+Q
Transform
View
Mesh
Scene
Import
Filter nodes
Scene Root
4way
StaticBody
CollisionShape
Thill
StaticBody
CollisionShape
Tplain
Inspector
Node
4way
MeshInstance
Mesh
Skeleton
Material
GeometryInstance
Output
Debugger
Audio
Animation

You can find an already exported MeshLibrary in the project named "MeshLibrary.tres".

Using GridMap

Create a new scene and add a GridMap node. Add the mesh library by dragging the resource file from the FileSystem dock and dropping it in the "Theme" property in the Inspector.


Godot Engine - Gridmaps Demo - GridMap_Demo.tscn (*)
Scene Project Debug Editor Help
2D 3D Script AssetLib
FileSystem
res://
Favorites:
res://
assets
dae
assets
default_env.tres
icon.png
MeshLibrary.tres
MeshLibrary_Source.tscn
GridMap_Demo.tscn
MeshLibrary_Source
GridMap_Demo(*)
Transform View
Floor: 0
Grid Map
[Perspective]
4way
Thill
Tplain
Tsplit
bridgelane
cap
capflat
cornerbevel
cornersharp
diplane
Output Debugger Audio Animation
Scene Import
Filter nodes
GridMap_Demo
Inspector Node
GridMap_Demo
GridMap

Theme MeshLib
Cell
Size (2, 2, 2)
Octant Size 8
Center X ☑ On
Center Y ☐ On
Center Z ☑ On
Scale 1
Collision
Layer
Mask

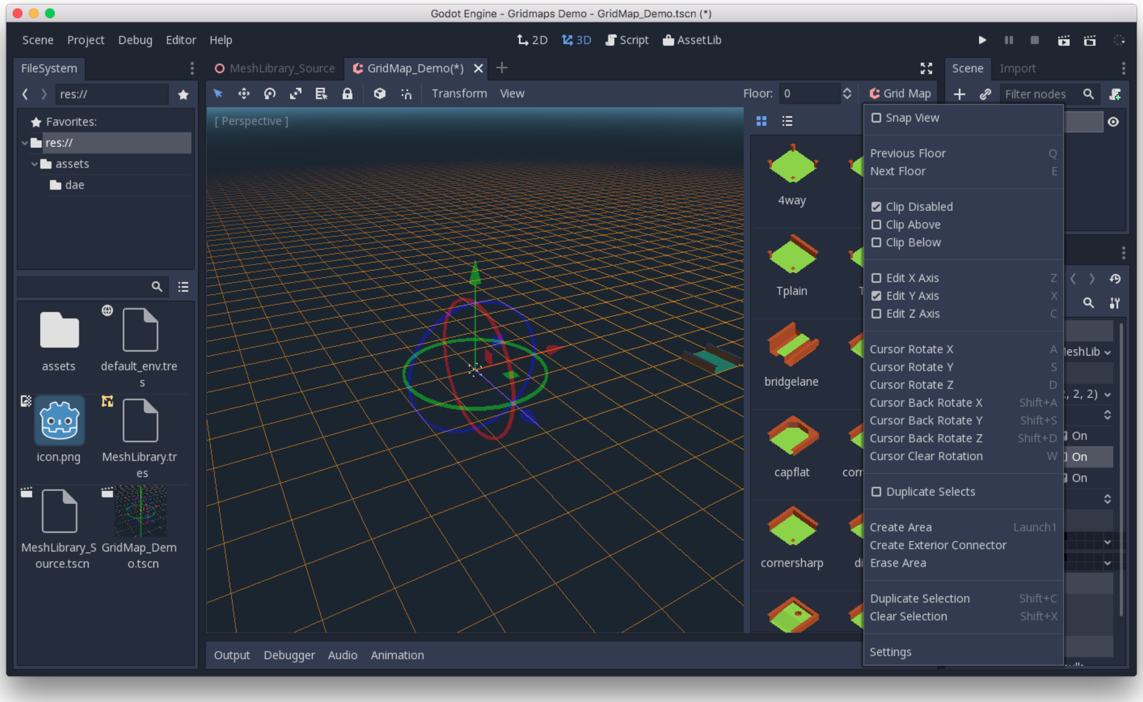
Spatial
Transform
Visibility
Node

The "Cell/Size" property should be set to the size of your meshes. You can leave it at the default value for the demo. Set the "Center Y" property to "Off".

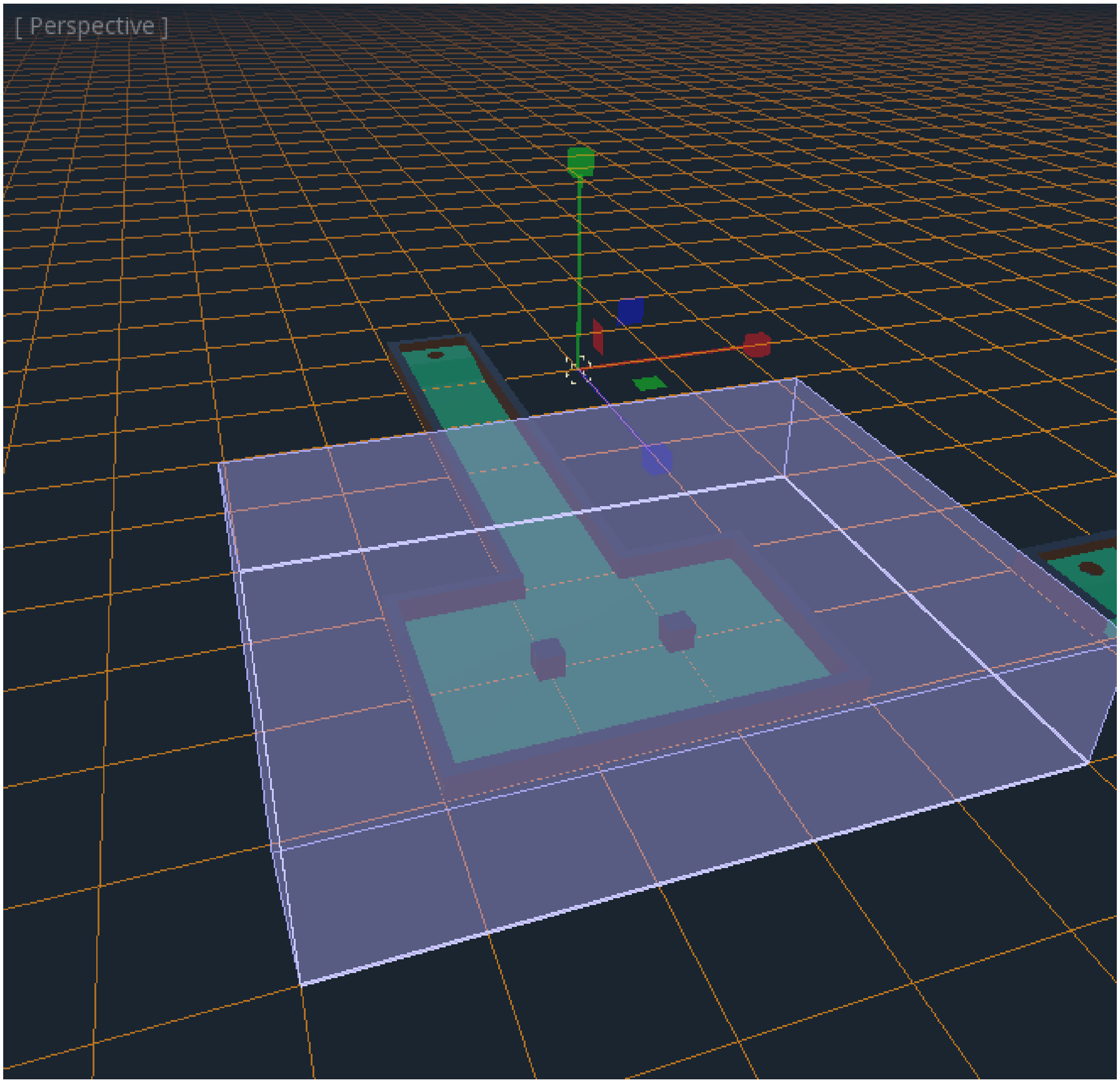
Now you can start designing the level by choosing a tile from the palette and placing it with Left-Click in the editor window. Use Right-click to remove a tile.

Use the arrows next to the "GridMap" menu to change the floor that you are working on.

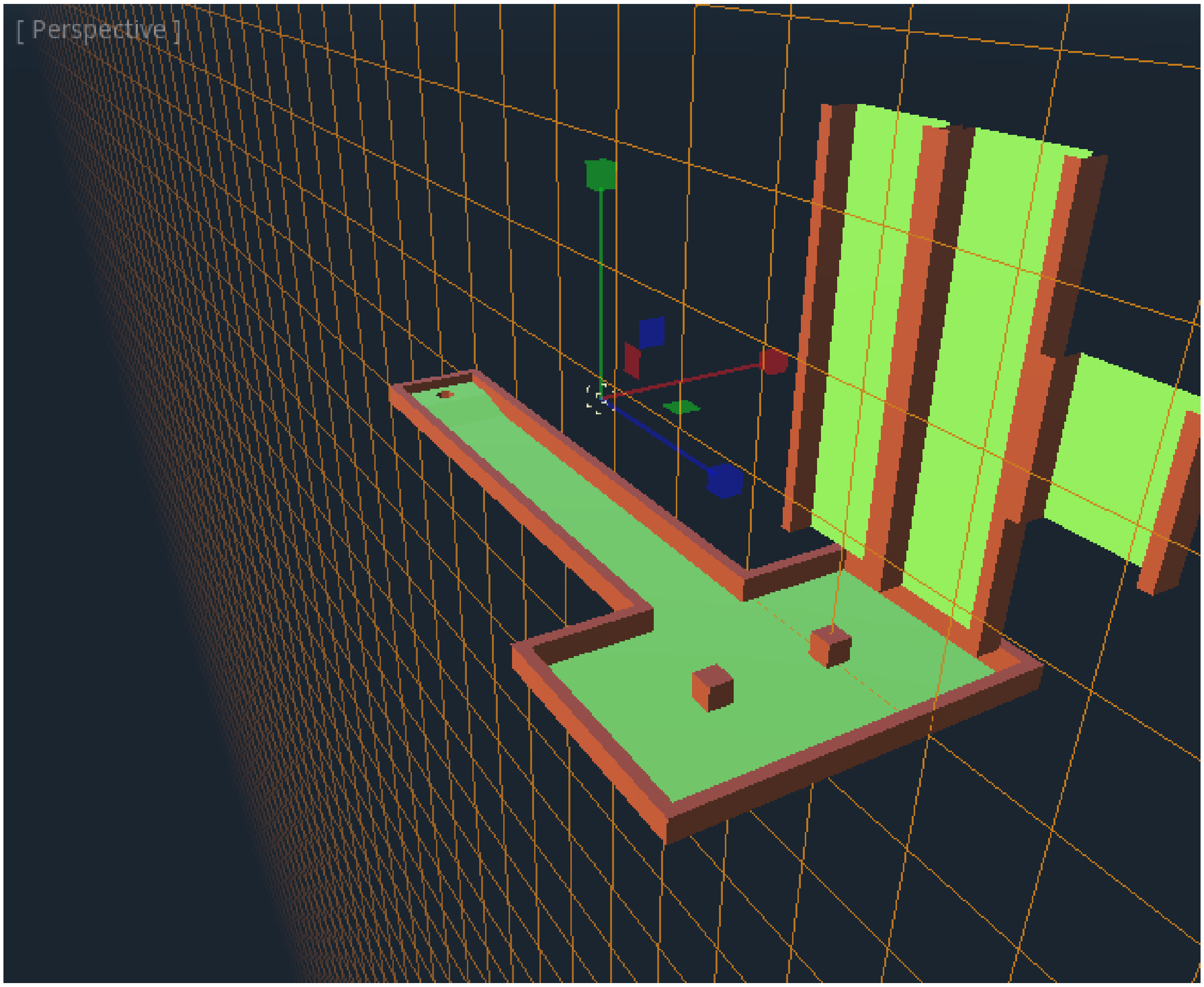
Click on the "GridMap" menu to see options and shortcuts. For example, pressing `s` rotates a tile around the y-axis.


Godot Engine - Gridmaps Demo - GridMap_Demo.tscn (*)
Scene Project Debug Editor Help
2D 3D Script AssetLib
FileSystem
MeshLibrary_Source
GridMap_Demo(*)
Transform View
Floor: 0
Grid Map
Scene Import
Filter nodes
res://
Favorites:
res://
assets
dae
[Perspective]
4way
Tplain
bridgelane
capflat
cornersharp
assets
default_env.tres
icon.png
MeshLibrary.tres
MeshLibrary_Source.tscn
GridMap_Demo.tscn
☐ Snap View
Previous Floor Q
Next Floor E
☑ Clip Disabled
☐ Clip Above
☐ Clip Below
☐ Edit X Axis Z
☑ Edit Y Axis X
☐ Edit Z Axis C
Cursor Rotate X A
Cursor Rotate Y S
Cursor Rotate Z D
Cursor Back Rotate X Shift+A
Cursor Back Rotate Y Shift+S
Cursor Back Rotate Z Shift+D
Cursor Clear Rotation W
☐ Duplicate Selects
Create Area Launch1
Create Exterior Connector
Erase Area
Duplicate Selection Shift+C
Clear Selection Shift+X
Settings
Output Debugger Audio Animation

Holding `Shift` and dragging with the left mouse button will draw a selection box. You can duplicate or clear the selected area using the respective menu options.


[Perspective]

In the menu, you can also change the axis you're drawing on, as well as shift the drawing plane higher or lower on its axis.


[Perspective]

Using GridMap in code

See GridMap for details on the node's methods and member variables.

Animation

This section of the tutorial covers using the two animation nodes in Godot and the animation editor.

- Introduction to the animation features
- Animation Track types
- Cutout animation
- 2D skeletons
- Using AnimationTree
- Playing videos
- Creating movies

See Importing 3D scenes for information on importing animations from a 3D model.

Introduction to the animation features

The AnimationPlayer node allows you to create anything from simple to complex animations.

In this guide you learn to:

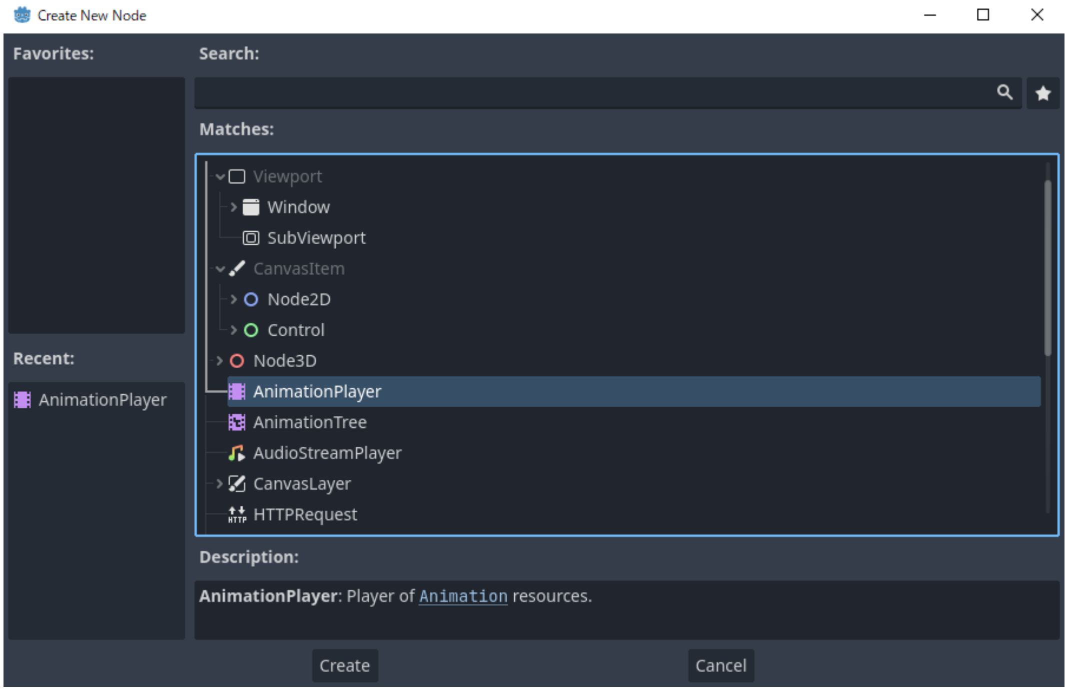
- Work with the Animation Panel
- Animate any property of any node
- Create a simple animation

In Godot, you can animate anything available in the Inspector, such as Node transforms, sprites, UI elements, particles, visibility and color of materials, and so on. You can also modify values of script variables and even call functions.

Create an AnimationPlayer node

To use the animation tools we first have to create an AnimationPlayer node.

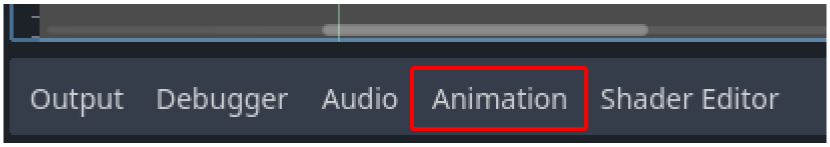
The AnimationPlayer node type is the data container for your animations. One AnimationPlayer node can hold multiple animations, which can automatically transition to one another.


Create New Node
Favorites:
Search:
Matches:
Viewport
Window
SubViewport
CanvasItem
Node2D
Control
Node3D
AnimationPlayer
AnimationTree
AudioStreamPlayer
CanvasLayer
HTTPRequest
Recent:
AnimationPlayer
Description:
Create
Cancel

AnimationPlayer: Player of Animation resources.

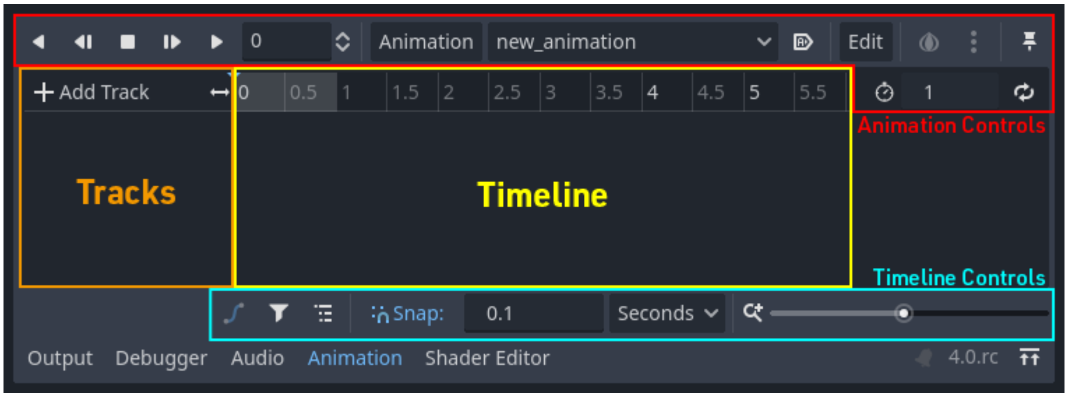
The AnimationPlayer node

After you create an AnimationPlayer node, click on it to open the Animation Panel at the bottom of the viewport.


Output
Debugger
Audio
Animation
Shader Editor

The animation panel position

The animation panel consists of four parts:


0
Animation
new_animation
Edit
Add Track
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5 5.5
1
Animation Controls
Tracks
Timeline
Timeline Controls
Snap:
0.1
Seconds
Output
Debugger
Audio
Animation
Shader Editor
4.0.rc

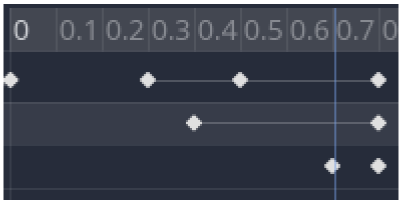
The animation panel

- Animation controls (i.e. add, load, save, and delete animations)
- The tracks listing
- The timeline with keyframes
- The timeline and track controls, where you can zoom the timeline and edit tracks, for example.

Computer animation relies on keyframes

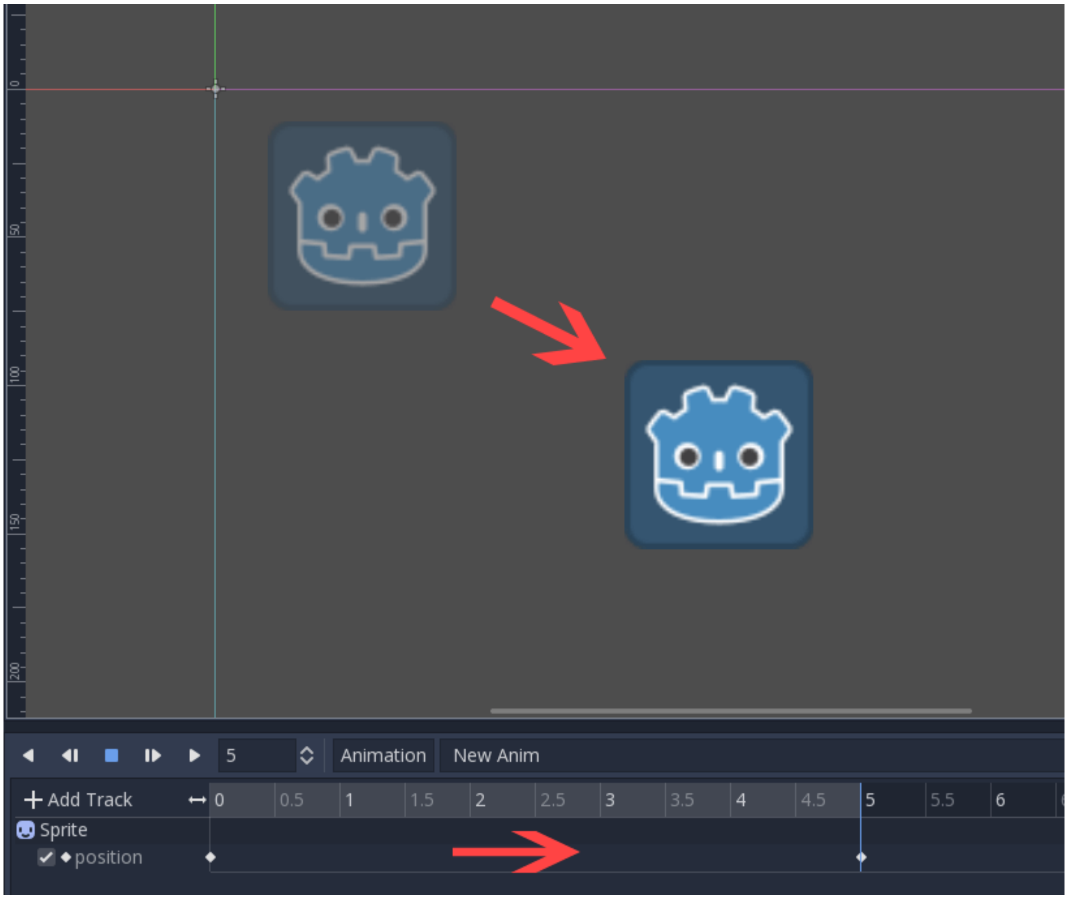
A keyframe defines the value of a property at a point in time.

Diamond shapes represent keyframes in the timeline. A line between two keyframes indicates that the value doesn't change between them.



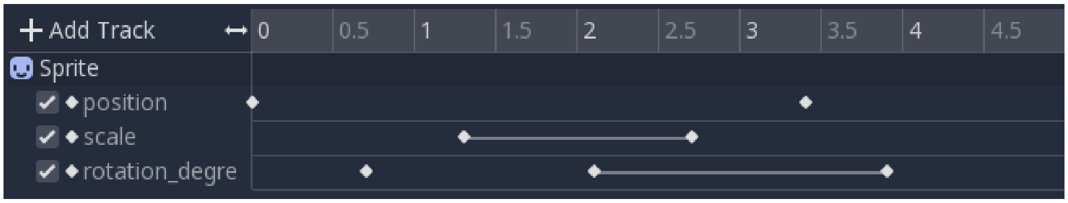
Keyframes in Godot

You set values of a node's properties and create animation keyframes for them. When the animation runs, the engine will interpolate the values between the keyframes, resulting in them gradually changing over time.


5
Animation
New Anim
Add Track
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
Sprite
position

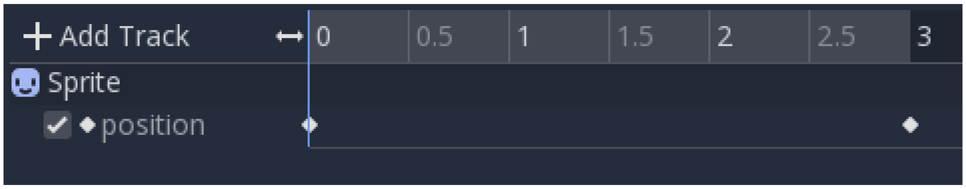
Two keyframes are all it takes to obtain a smooth motion

The timeline defines how long the animation will take. You can insert keyframes at various points, and change their timing.


Add Track
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Sprite
position
scale
rotation_degre

The timeline in the animation panel

Each line in the Animation Panel is an animation track that references a Normal or Transform property of a node. Each track stores a path to a node and its affected property. For example, the position track in the illustration refers to the `position` property of the Sprite2D node.



Example of Normal animation tracks

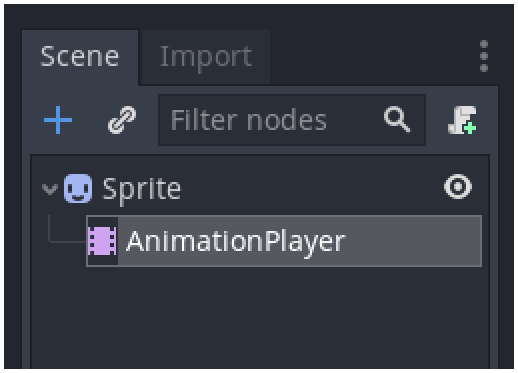
Tip

If you animate the wrong property, you can edit a track's path at any time by double-clicking on it and typing the new path. Play the animation using the "Play from beginning" button ▶ (or pressing `Shift + D` on keyboard) to see the changes instantly.

Tutorial: Creating a simple animation

Scene setup

For this tutorial, we'll create a Sprite node with an AnimationPlayer as its child. We will animate the sprite to move between two points on the screen.



Our scene setup

Warning

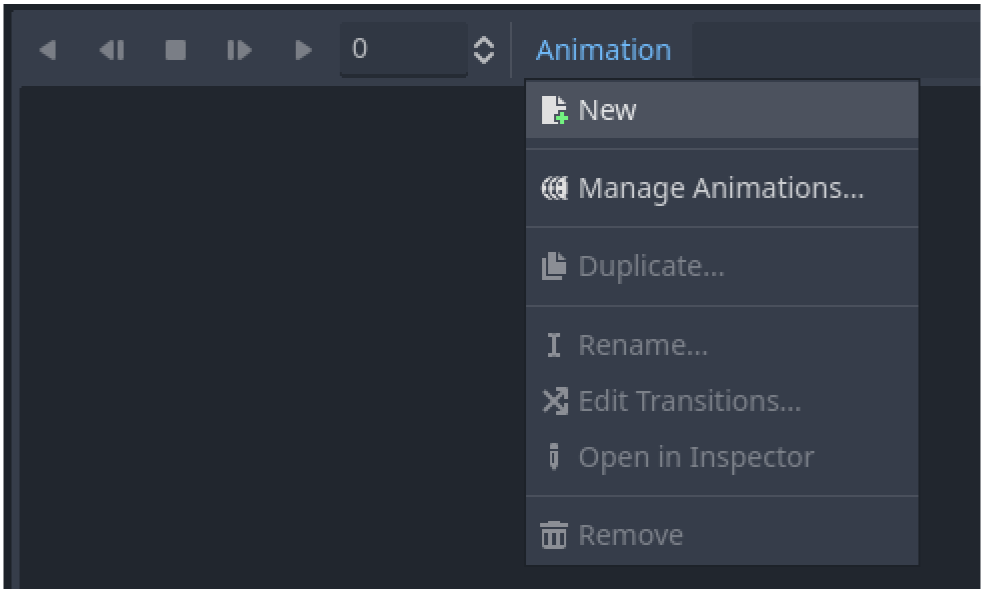
AnimationPlayer inherits from Node instead of Node2D or Node3D, which means that the child nodes will not inherit the transform from the parent nodes due to a bare Node being present in the hierarchy.

Therefore, it is not recommended to add nodes that have a 2D/3D transform as a child of an AnimationPlayer node.

The sprite holds an image texture. For this tutorial, select the Sprite2D node, click Texture in the Inspector, and then click Load. Select the default Godot icon for the sprite's texture.

Adding an animation

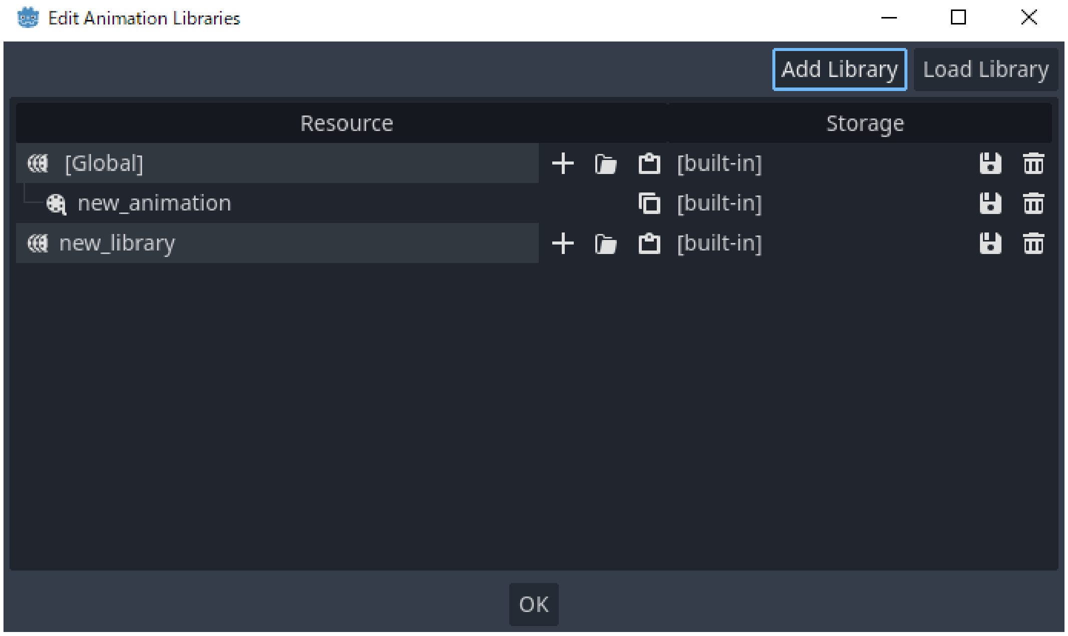
Select the AnimationPlayer node and click the "Animation" button in the animation editor. From the list, select "New" () to add a new animation. Enter a name for the animation in the dialog box.


0
Animation
New
Manage Animations...
Duplicate...
Rename...
Edit Transitions...
Open in Inspector
Remove

Add a new animation

Managing animation libraries

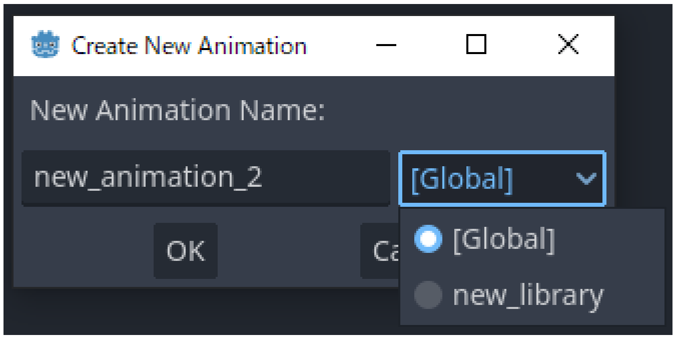
For reusability, the animation is registered in a list in the animation library resource. If you add an animation to AnimationPlayer without specifying any particular settings, the animation will be registered in the [Global] animation library that AnimationPlayer has by default.


Edit Animation Libraries
Add Library
Load Library
OK

Resource	Storage
[Global]	[built-in]
new_animation	[built-in]
new_library	[built-in]

Manage animations

If there are multiple animation libraries and you try to add an animation, a dialog box will appear with options.



New Animation Name:

new_animation_2

[Global]

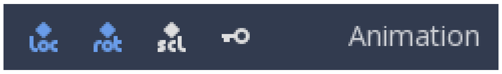
● [Global]

○ new_library

Add a new animation with library option

Adding a track

To add a new track for our sprite, select it and take a look at the toolbar:

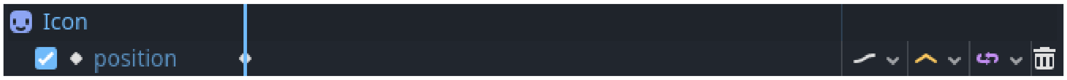


Convenience buttons

These switches and buttons allow you to add keyframes for the selected node's location, rotation, and scale. Since we are only animating the sprite's position, make sure that only the location switch is selected. The selected switches are blue.

Click on the key button to create the first keyframe. Since we don't have a track set up for the Position property yet, Godot will offer to create it for us. Click **Create**.

Godot will create a new track and insert our first keyframe at the beginning of the timeline:

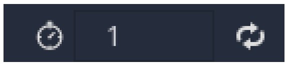


The sprite track

The second keyframe

We need to set our sprite's end location and how long it will take for it to get there.

Let's say we want it to take two seconds to move between the points. By default, the animation is set to last only one second, so change the animation length to 2 in the controls on the right side of the animation panel's timeline header.



Animation length

Now, move the sprite right, to its final position. You can use the *Move tool* in the toolbar or set the *Position*'s X value in the *Inspector*.

Click on the timeline header near the two-second mark in the animation panel and then click the key button in the toolbar to create the second keyframe.

Run the animation

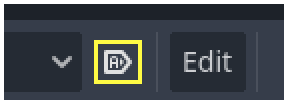
Click on the "Play from beginning" (▶) button.

Yay! Our animation runs:

The animation

Autoplay on load

You can make it so an animation plays automatically when the AnimationPlayer nodes scene starts, or joins another scene. To do this click the "Autoplay on load" button in the animation editor, it's right next to the edit button.

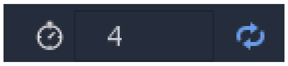


The icon for it will also appear in front of the name of the animation, so you can easily identify which one is the autoplay animation.

Back and forth

Godot has an interesting feature that we can use in animations. When Animation Looping is set but there's no keyframe specified at the end of the animation, the first keyframe is also the last.

This means we can extend the animation length to four seconds now, and Godot will also calculate the frames from the last keyframe to the first, moving our sprite back and forth.

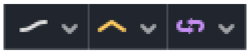


Animation loop

You can change this behavior by changing the track's loop mode. This is covered in the next chapter.

Track settings

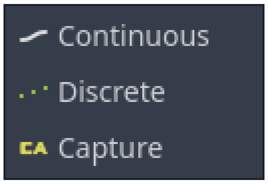
Each property track has a settings panel at the end, where you can set its update mode, track interpolation, and loop mode.



Track settings

The update mode of a track tells Godot when to update the property values. This can be:

- **Continuous:** Update the property on each frame
- **Discrete:** Only update the property on keyframes
- **Capture:** if the first keyframe's time is greater than `0.0`, the current value of the property will be remembered and will be blended with the first animation key. For example, you could use the Capture mode to move a node that's located anywhere to a specific location.

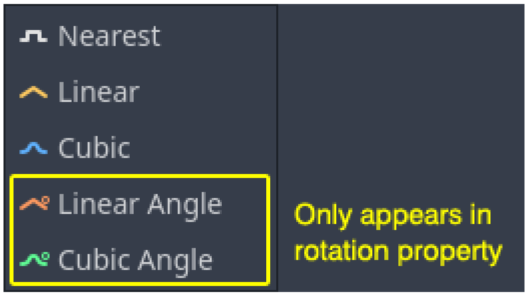


Track mode

You will usually use "Continuous" mode. The other types are used to script complex animations.

Track interpolation tells Godot how to calculate the frame values between keyframes. These interpolation modes are supported:

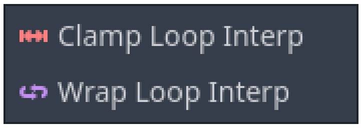
- Nearest: Set the nearest keyframe value
- Linear: Set the value based on a linear function calculation between the two keyframes
- Cubic: Set the value based on a cubic function calculation between the two keyframes
- Linear Angle (Only appears in rotation property): Linear mode with shortest path rotation
- Cubic Angle (Only appears in rotation property): Cubic mode with shortest path rotation


Nearest
Linear
Cubic
Linear Angle
Cubic Angle
Only appears in rotation property

Track interpolation

With Cubic interpolation, animation is slower at keyframes and faster between them, which leads to more natural movement. Cubic interpolation is commonly used for character animation. Linear interpolation animates changes at a fixed pace, resulting in a more robotic effect.

Godot supports two loop modes, which affect the animation when it's set to loop:


Clamp Loop Interp
Wrap Loop Interp

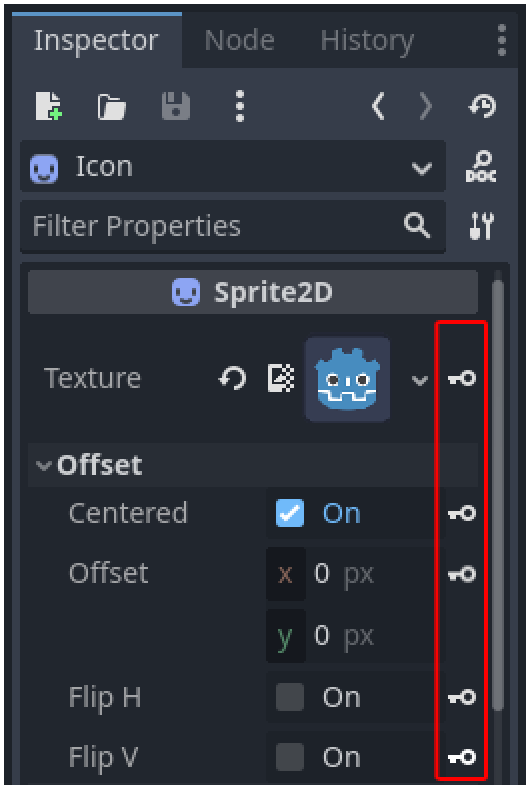
Loop modes

- Clamp loop interpolation: When this is selected, the animation stops after the last keyframe for this track. When the first keyframe is reached again, the animation will reset to its values.
- Wrap loop interpolation: When this is selected, Godot calculates the animation after the last keyframe to reach the values of the first keyframe again.

Keyframes for other properties

Godot's animation system isn't restricted to position, rotation, and scale. You can animate any property.

If you select your sprite while the animation panel is visible, Godot will display a small keyframe button in the *Inspector* for each of the sprite's properties. Click on one of these buttons to add a track and keyframe to the current animation.


Inspector
Node
History
Icon
Filter Properties
Sprite2D
Texture
Offset
Centered
Offset
Flip H
Flip V

Centered ☑ On

Offset x 0 px y 0 px

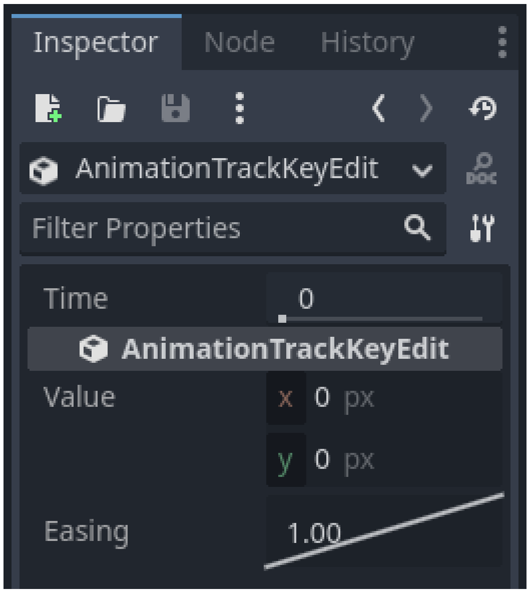
Flip H ☐ On

Flip V ☐ On

Keyframes for other properties

Edit keyframes

You can click on a keyframe in the animation timeline to display and edit its value in the *Inspector*.



Time 0

AnimationTrackKeyEdit

Value x 0 px, y 0 px

Easing 1.00

Keyframe editor editing a key

You can also edit the easing value for a keyframe here by clicking and dragging its easing curve. This tells Godot how to interpolate the animated property when it reaches this keyframe.

You can tweak your animations this way until the movement "looks right."

Using RESET tracks

You can set up a special *RESET* animation to contain the "default pose". This is used to ensure that the default pose is restored when you save the scene and open it again in the editor.

For existing tracks, you can add an animation called "RESET" (case-sensitive), then add tracks for each property that you want to reset. The only keyframe should be at time 0, and give it the desired default value for each track.

If AnimationPlayer's **Reset On Save** property is set to `true`, the scene will be saved with the effects of the reset animation applied (as if it had been seeked to time `0.0`). This only affects the saved file – the property tracks in the editor stay where they were.

If you want to reset the tracks in the editor, select the AnimationPlayer node, open the **Animation** bottom panel then choose **Apply Reset** in the animation editor's **Edit** dropdown menu.

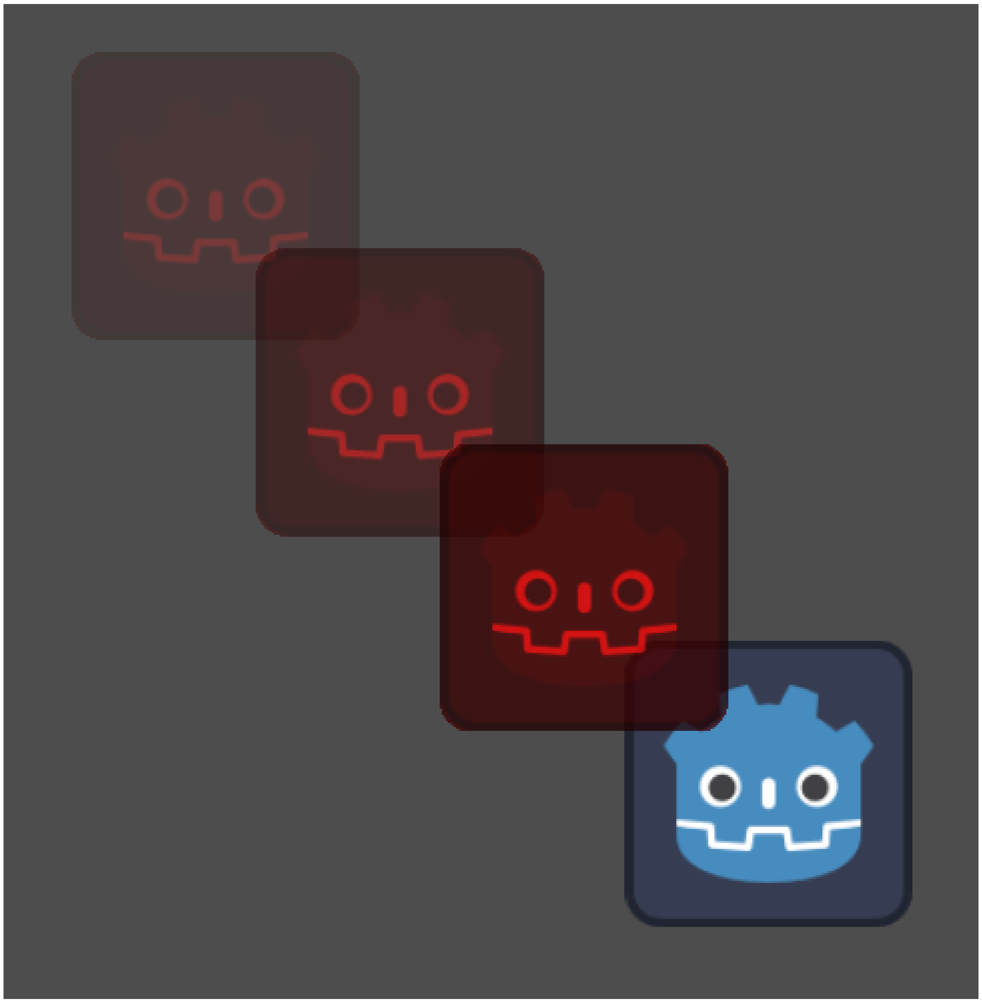
When using the keyframe icon next to a property in the inspector the editor will ask you to automatically create a RESET track.

Note

RESET tracks are also used as reference values for blending. See also For better blending.

Onion Skinning

Godot's animation editor allows you use onion skinning while creating an animation. To turn this feature on click on the onion icon in the top right of the animation editor. Now there will be transparent red copies of what is being animated in its previous positions in the animation.



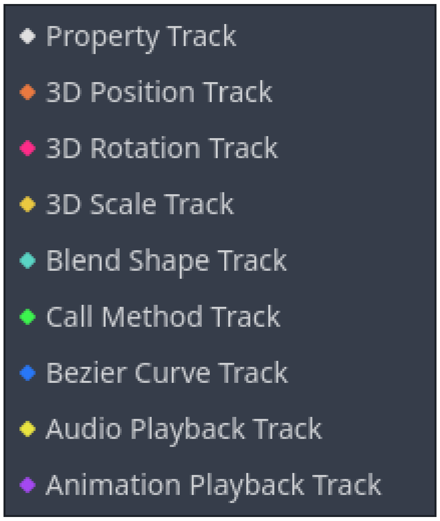
The three dots button next to the onion skinning button opens a dropdown menu that lets you adjust how it works, including the ability to use onion skinning for future frames.

Animation Track types

This page gives an overview of the track types available for Godot's animation player node on top of the default property tracks.

See also

We assume you already read Introduction to the animation features, which covers the basics, including property tracks.


Property Track
3D Position Track
3D Rotation Track
3D Scale Track
Blend Shape Track
Call Method Track
Bezier Curve Track
Audio Playback Track
Animation Playback Track

Property Track

The most basic track type. See Introduction to the animation features.

Position 3D / Rotation 3D / Scale 3D Track

These 3D transform tracks control the location, rotation, and scale of a 3D object. They make it easier to animate a 3D object's transform compared to using regular property tracks.

It is designed for animations imported from external 3D models and can reduce resource capacity through compression.

Blend Shape Track

A blend shape track is optimized for animating blend shape in MeshInstance3D.

It is designed for animations imported from external 3D models and can reduce resource capacity through compression.

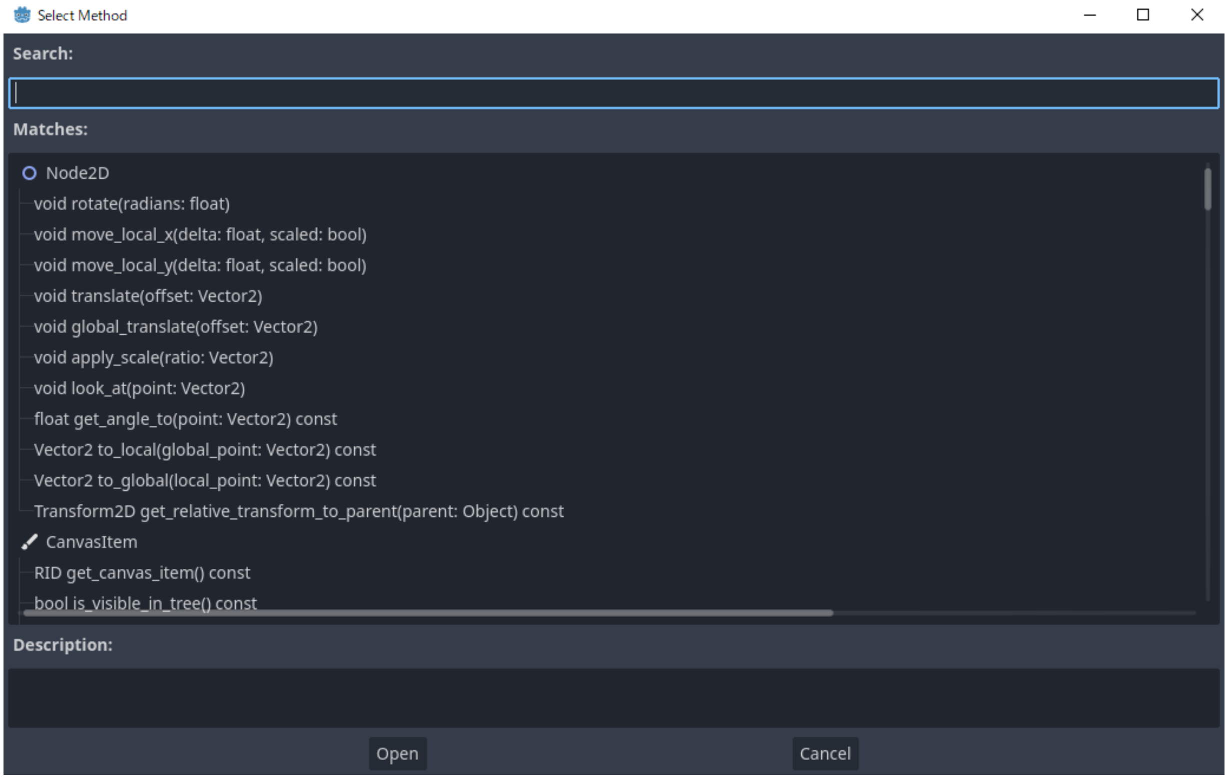
Call Method Track

A call method track allow you to call a function at a precise time from within an animation. For example, you can call `queue_free()` to delete a node at the end of a death animation.

Note

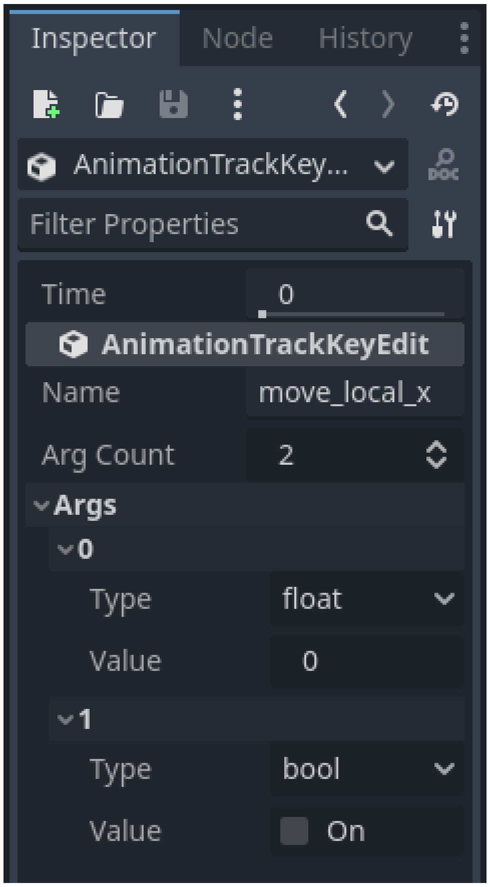
The events placed on the call method track are not executed when the animation is previewed in the editor for safety.

To create such a track in the editor, click "Add Track -> Call Method Track." Then, a window opens and lets you select the node to associate with the track. To call one of the node's methods, right-click the timeline and select "Insert Key". A window opens with a list of available methods. Double-click one to finish creating the keyframe.



Node2D
void rotate(radians: float)
void move_local_x(delta: float, scaled: bool)
void move_local_y(delta: float, scaled: bool)
void translate(offset: Vector2)
void global_translate(offset: Vector2)
void apply_scale(ratio: Vector2)
void look_at(point: Vector2)
float get_angle_to(point: Vector2) const
Vector2 to_local(global_point: Vector2) const
Vector2 to_global(local_point: Vector2) const
Transform2D get_relative_transform_to_parent(parent: Object) const
CanvasItem
RID get_canvas_item() const
bool is_visible_in_tree() const

To change the method call or its arguments, click on the key and head to the inspector dock. There, you can change the method to call. If you expand the "Args" section, you will see a list of arguments you can edit.


Inspector
Node
History
AnimationTrackKey...
Filter Properties
AnimationTrackKeyEdit

Time 0

Name move_local_x

Arg Count 2

Args

0

Type float

Value 0

1

Type bool

Value ☐ On

To create such a track through code, pass a dictionary that contains the target method's name and parameters as the Variant for `key` in `Animation.track_insert_key()`. The keys and their expected values are as follows:

Key	Value
`"method"`	The name of the method as a `String`
`"args"`	The arguments to pass to the function as an `Array`

GDScript

```
# Create a call method track.
func create_method_animation_track():
    # Get or create the animation the target method will be
called from.
    var animation = $AnimationPlayer.get_animation("idle")
    # Get or create the target method's animation track.
    var track_index =
animation.add_track(Animation.TYPE_METHOD)
    # Make the arguments for the target method jump().
    var jump_velocity = -400.0
    var multiplier = randf_range(.8, 1.2)
    # Get or create a dictionary with the target method's name
and arguments.
    var method_dictionary = {
        "method": "jump",
        "args": [jump_velocity, multiplier],
    }

    # Set scene-tree path to node with target method.
    animation.track_set_path(track_index, ".")
    # Add the dictionary as the animation method track's key.
    animation.track_insert_key(track_index, 0.6,
method_dictionary, 0)


# The target method that will be called from the animation.
func jump(jump_velocity, multiplier):
    velocity.y = jump_velocity * multiplier
```

C#

```
// Create a call method track.
public void CreateAnimationTrack()
{
    // Get reference to the AnimationPlayer.
    var animationPlayer = GetNode<AnimationPlayer>
("AnimationPlayer");
    // Get or create the animation the target method will be
called from.
    var animation = animationPlayer.GetAnimation("idle");
    // Get or create the target method's animation track.
    var trackIndex =
animation.AddTrack(Animation.TrackType.Method);
```

```
    // Make the arguments for the target method Jump().
    var jumpVelocity = -400.0;
    var multiplier = GD.RandRange(.8, 1.2);
    // Get or create a dictionary with the target method's name and arguments.
    var methodDictionary = new Godot.Collections.Dictionary
    {
        { "method", MethodName.Jump },
        { "args", new Godot.Collections.Array { jumpVelocity, multiplier } }
    };

    // Set scene-tree path to node with target method.
    animation.TrackSetPath(trackIndex, ".");
    // Add the dictionary as the animation method track's key.
    animation.TrackInsertKey(trackIndex, 0.6, methodDictionary, 0);
}


// The target method that will be called from the animation.
private void Jump(float jumpVelocity, float multiplier)
{
    Velocity = new Vector2(Velocity.X, jumpVelocity * multiplier);
}
```

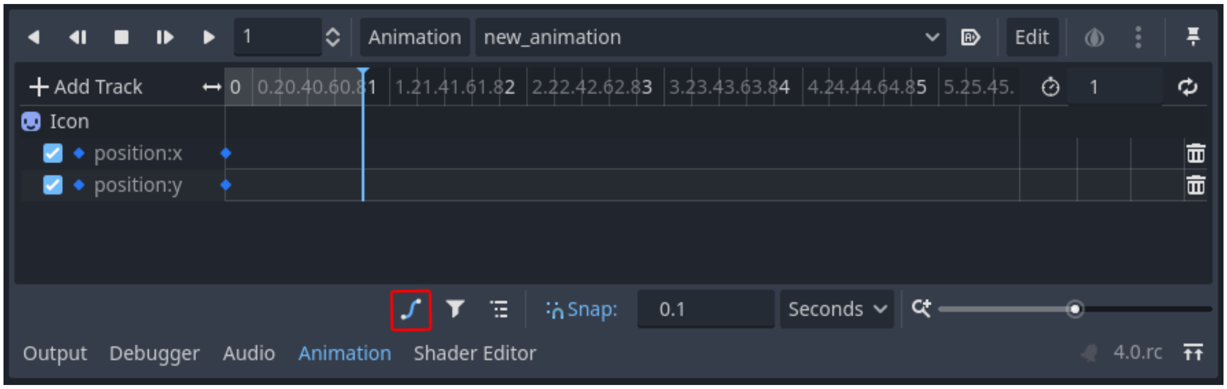
Bezier Curve Track

A bezier curve track is similar to a property track, except it allows you to animate a property's value using a bezier curve.

Note

Bezier curve track and property track cannot be blended in AnimationPlayer and AnimationTree.

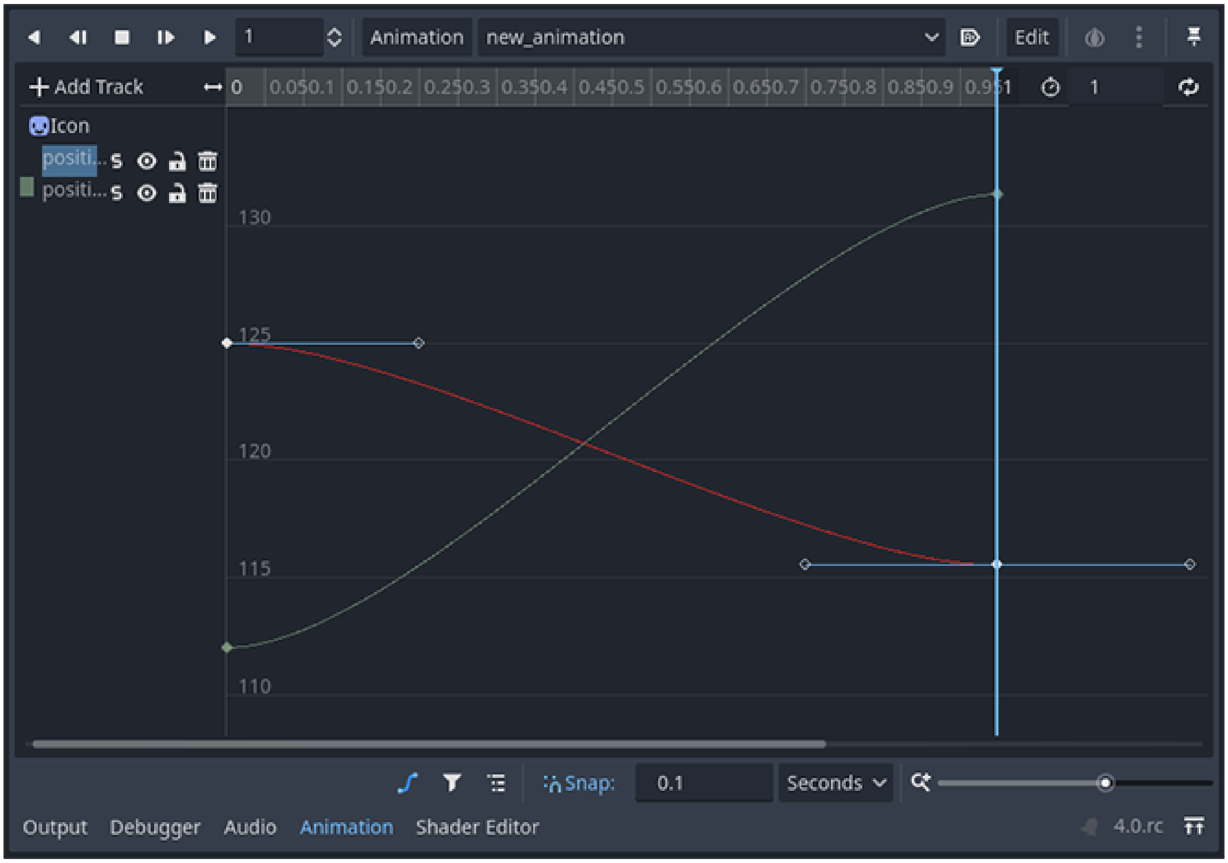
To create one, click "Add Track -> Bezier Curve Track". As with property tracks, you need to select a node and a property to animate. To open the bezier curve editor, click the curve icon to the right of the animation track.


Animation
new_animation
Edit
Add Track
Icon
position:x
position:y
Snap:
0.1
Seconds
Output
Debugger
Audio
Animation
Shader Editor
4.0.rc

In the editor, keys are represented by filled diamonds and the outlined diamonds connected to them by a line control curve's shape.

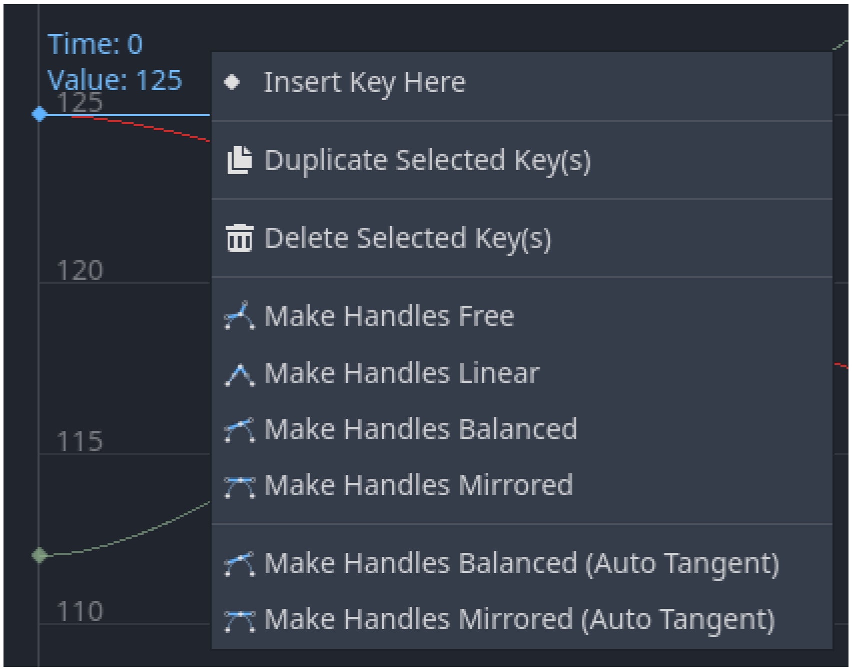
Tip

For better precision while manually working with curves, you might want to alter the zoom levels of the editor. The slider on the bottom right of the editor can be used to zoom in and out on the time axis, you can also do that with `Ctrl + Shift + Mouse wheel`. Using `Ctrl + Alt + Mouse wheel` will zoom in and out on the Y axis


1
Animation
new_animation
Edit
Add Track
0 0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5 0.55 0.6 0.65 0.7 0.75 0.8 0.85 0.9 0.95 1
1
Icon
positi...
positi...
130
125
120
115
110
Snap:
0.1
Seconds
Output
Debugger
Audio
Animation
Shader Editor
4.0.rc

In the right click panel of the editor, you can select the handle mode:

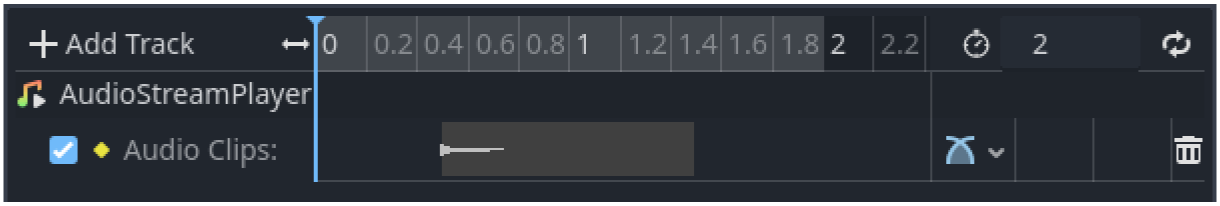
- Free: Allows you to orient a manipulator in any direction without affecting the other's position.
- Linear: Does not allow rotation of the manipulator and draws a linear graph.
- Balanced: Makes it so manipulators rotate together, but the distance between the key and a manipulator is not mirrored.
- Mirrored: Makes the position of one manipulator perfectly mirror the other, including their distance to the key.


Time: 0
Value: 125
125
120
115
110
Insert Key Here
Duplicate Selected Key(s)
Delete Selected Key(s)
Make Handles Free
Make Handles Linear
Make Handles Balanced
Make Handles Mirrored
Make Handles Balanced (Auto Tangent)
Make Handles Mirrored (Auto Tangent)

Audio Playback Track

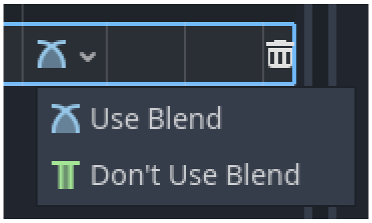
If you want to create an animation with audio, you need to create an audio playback track. To create one, your scene must have either an AudioStreamPlayer, AudioStreamPlayer2D, or AudioStreamPlayer3D node. When creating the track, you must select one of those nodes.

To play a sound in your animation, drag and drop an audio file from the file system dock onto the animation track. You should see the waveform of your audio file in the track.


Add Track
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2
AudioStreamPlayer
Audio Clips:

To remove a sound from the animation, you can right-click it and select "Delete Key(s)" or click on it and press the `Del` key.

The blend mode allows you to choose whether or not to adjust the audio volume when blending in the AnimationTree.



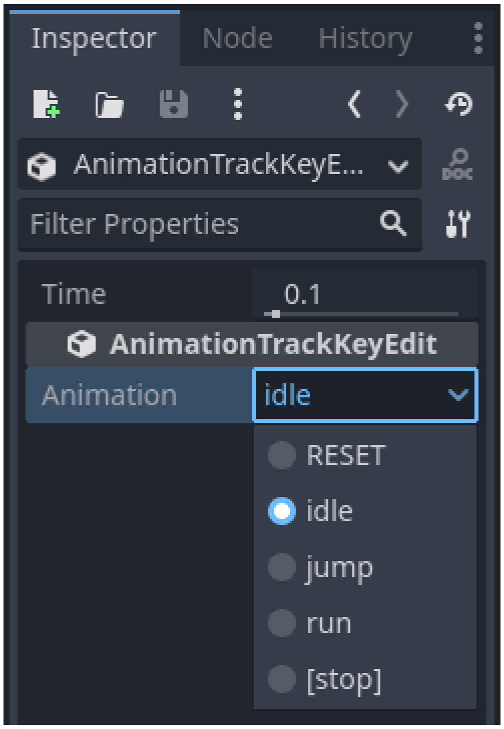
Animation Playback Track

Animation playback tracks allow you to sequence the animations of other animation player nodes in a scene. For example, you can use it to animate several characters in a cut-scene.

To create an animation playback track, select "New Track -> Animation Playback Track."

Then, select the animation player you want to associate with the track.

To add an animation to the track, right-click on it and insert a key. Select the key you just created to select an animation in the inspector dock.


Inspector
Node
History
AnimationTrackKeyE...
Filter Properties
AnimationTrackKeyEdit
RESET
idle
jump
run
[stop]

Time 0.1

Animation idle

If an animation is already playing and you want to stop it early, you can create a key and have it set to *[STOP]* in the inspector.

Note

If you instanced a scene that contains an animation player into your scene, you need to enable "Editable Children" in the scene tree to access its animation player. Also, an animation player cannot reference itself.

Cutout animation

What is it?

Traditionally, cutout animation [https://en.wikipedia.org/wiki/Cutout_animation] is a type of stop motion animation [https://en.wikipedia.org/wiki/Stop_motion] in which pieces of paper (or other thin material) are cut into special shapes and arranged in two-dimensional representations of characters and objects. Characters' bodies are usually made out of several pieces. The pieces are arranged and photographed once for each frame of the film. The animator moves and rotates the parts in small increments between each shot to create the illusion of movement when the images are played back quickly in sequence.

Simulations of cutout animation can now be created using software as seen in South Park [https://en.wikipedia.org/wiki/South_Park] and Jake and the Never Land Pirates [https://en.wikipedia.org/wiki/Jake_and_the_Never_Land_Pirates].

In video games, this technique has also become popular. Examples of this are Paper Mario [https://en.wikipedia.org/wiki/Super_Paper_Mario] or Rayman Origins [https://en.wikipedia.org/wiki/Rayman_Origins] .

Cutout animation in Godot

Godot provides tools for working with cutout rigs, and is ideal for the workflow:

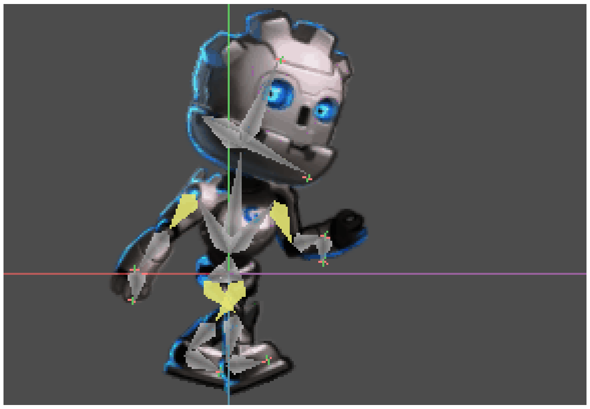
- **The animation system is fully integrated with the engine**: This means animations can control much more than just motion of objects. Textures, sprite sizes, pivots, opacity, color modulation, and more, can all be animated and blended.

- **Combine animation styles**: AnimatedSprite2D allows traditional cel animation to be used alongside cutout animation. In cel animation different animation frames use entirely different drawings rather than the same pieces positioned differently. In an otherwise cutout-based animation, cel animation can be used selectively for complex parts such as hands, feet, changing facial expressions, etc.
- **Custom Shaped Elements**: Custom shapes can be created with Polygon2D allowing UV animation, deformations, etc.
- **Particle Systems**: A cutout animation rig can be combined with particle systems. This can be useful for magic effects, jetpacks, etc.
- **Custom Colliders**: Set colliders and influence areas in different parts of the skeletons, great for bosses and fighting games.
- **Animation Tree**: Allows complex combinations and blending between several animations, the same way it works in 3D.

And much more!

Making of GBot

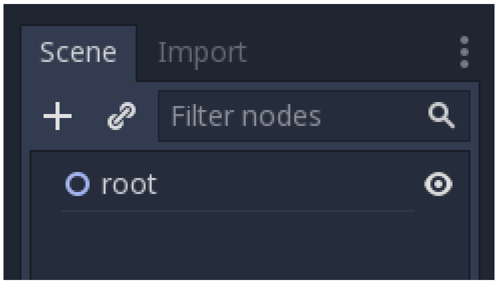
For this tutorial, we will use as demo content the pieces of the GBot [https://www.youtube.com/watch?v=S13FrWuBMx4&list=UUckpus81gNin1aV8WSffRKw] character, created by Andreas Esau.



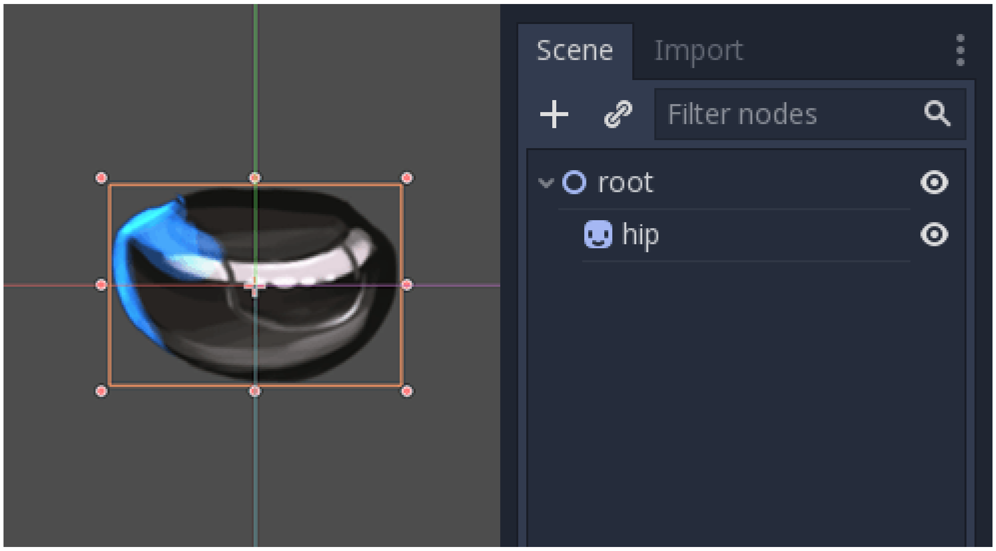
Get your assets: [cutout_animation_assets.zip](https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/cutout_animation_assets.zip) [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/cutout_animation_assets.zip].

Setting up the rig

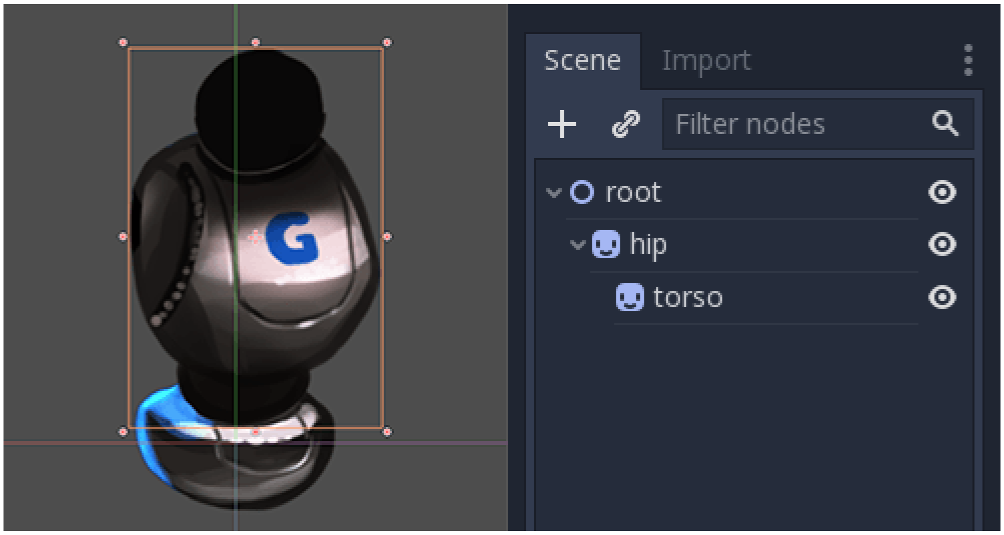
Create an empty Node2D as root of the scene, we will work under it:



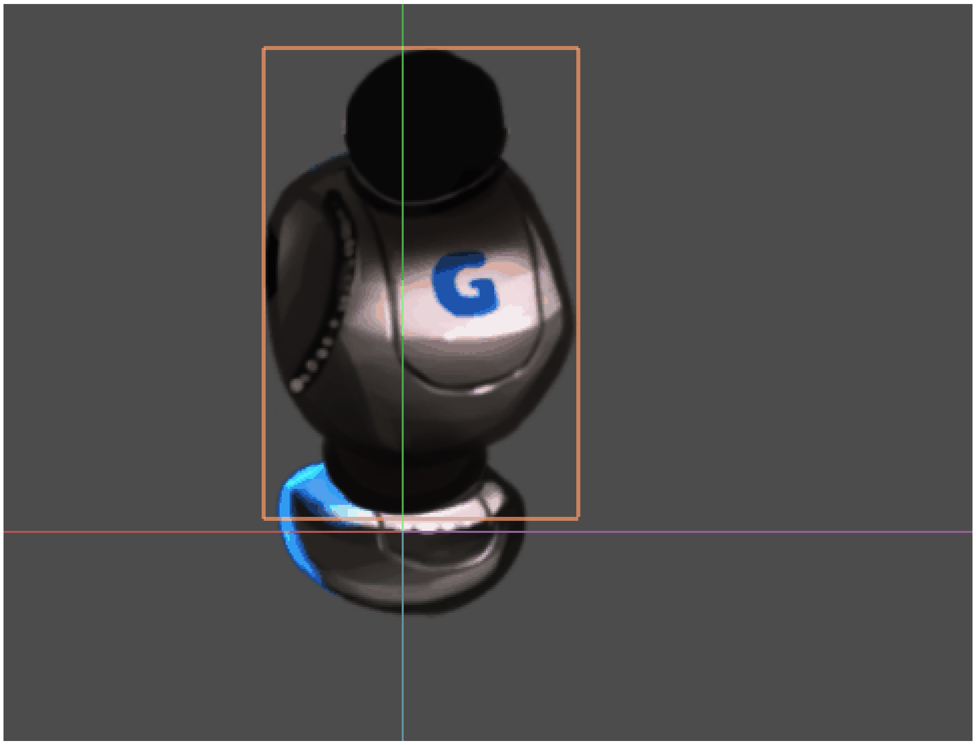
The first node of the model is the hip. Generally, both in 2D and 3D, the hip is the root of the skeleton. This makes it easier to animate:


Scene
Import
Filter nodes
root
hip

Next will be the torso. The torso needs to be a child of the hip, so create a child sprite and load the torso texture, later accommodate it properly:

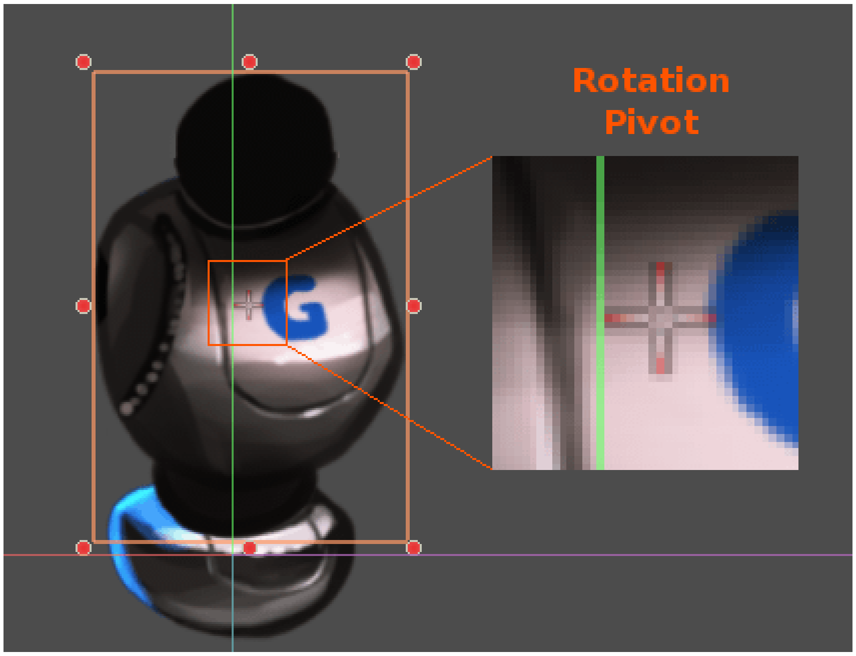

Scene
Import
Filter nodes
root
hip
torso

This looks good. Let's see if our hierarchy works as a skeleton by rotating the torso. We can do this be pressing `E` to enter rotate mode, and dragging with the left mouse button. To exit rotate mode hit `ESC`.



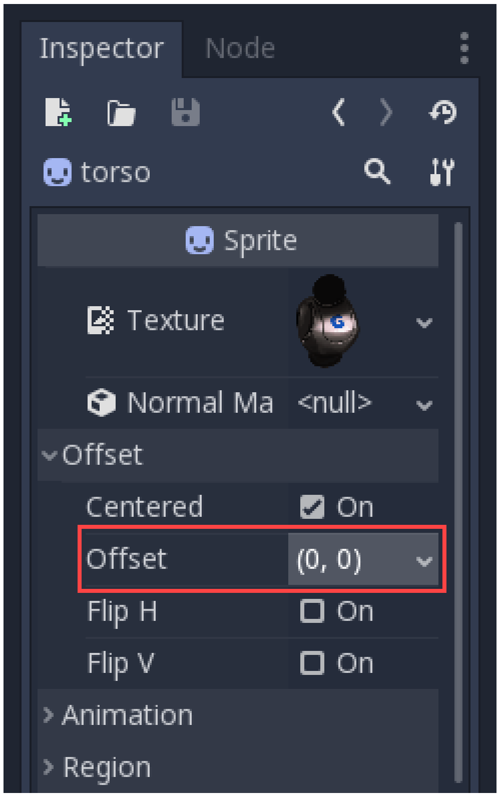
The rotation pivot is wrong and needs to be adjusted.

This small cross in the middle of the Sprite2D is the rotation pivot:


Rotation
Pivot

Adjusting the pivot

The pivot can be adjusted by changing the *offset* property in the Sprite2D:


Inspector
Node
torso
Sprite
Texture
Offset
Animation
Region

Normal Ma <null>

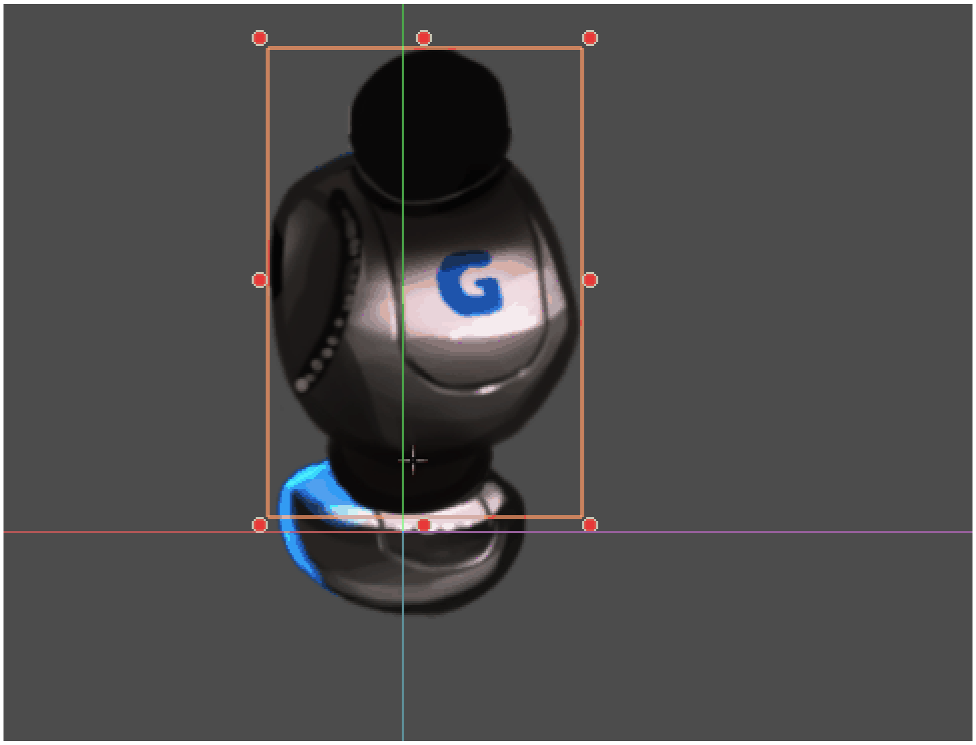
Centered ☑ On

Offset (0, 0)

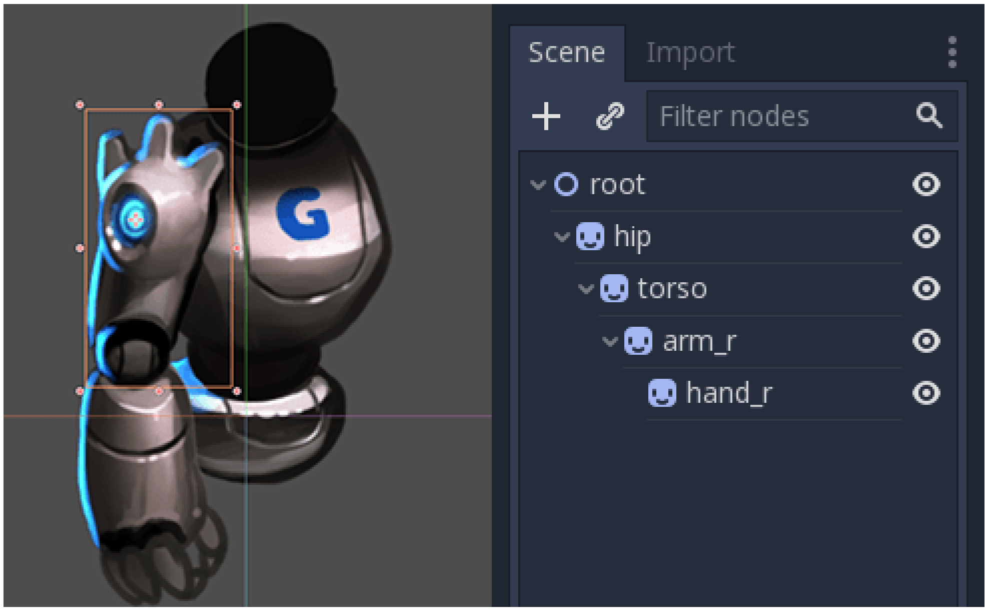
Flip H ☐ On

Flip V ☐ On

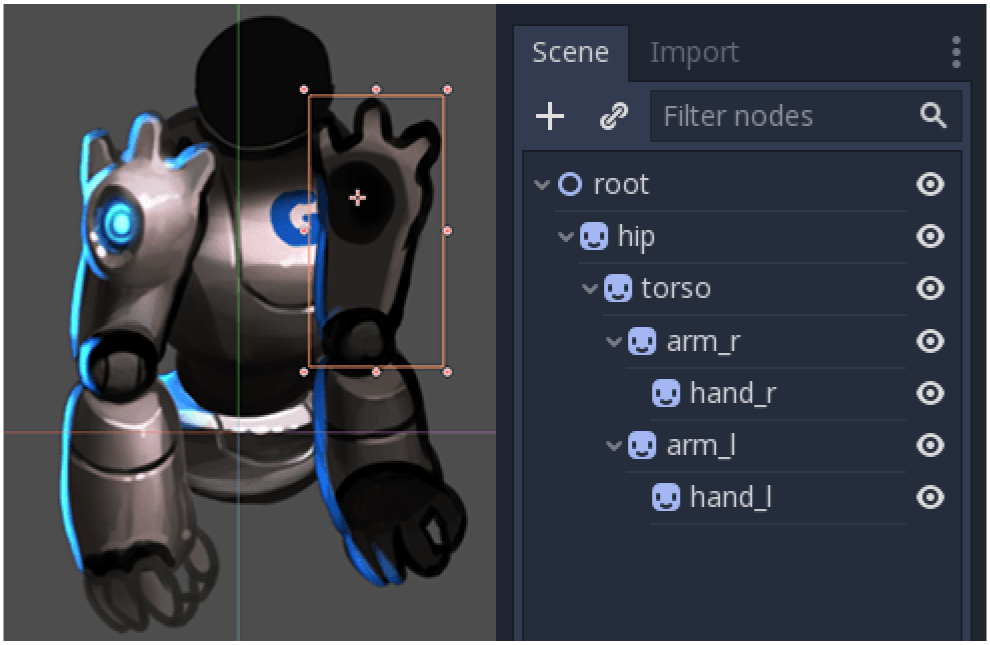
The pivot can also be adjusted *visually*. While hovering over the desired pivot point, press `v` to move the pivot there for the selected Sprite2D. There is also a tool in the tool bar that has a similar function.



Continue adding body pieces, starting with the right arm. Make sure to put each sprite in its correct place in the hierarchy, so its rotations and translations are relative to its parent:


Scene
Import
Filter nodes
root
hip
torso
arm_r
hand_r

With the left arm there's a problem. In 2D, child nodes appear in front of their parents:



We want the left arm to appear *behind* the hip and the torso. We could move the left arm nodes behind the hip (above the hip node in the scene hierarchy), but then the left arm is no longer in its proper place in the hierarchy. This means it wouldn't be affected by the movement of the torso. We'll fix this problem with `RemoteTransform2D` nodes.

Note

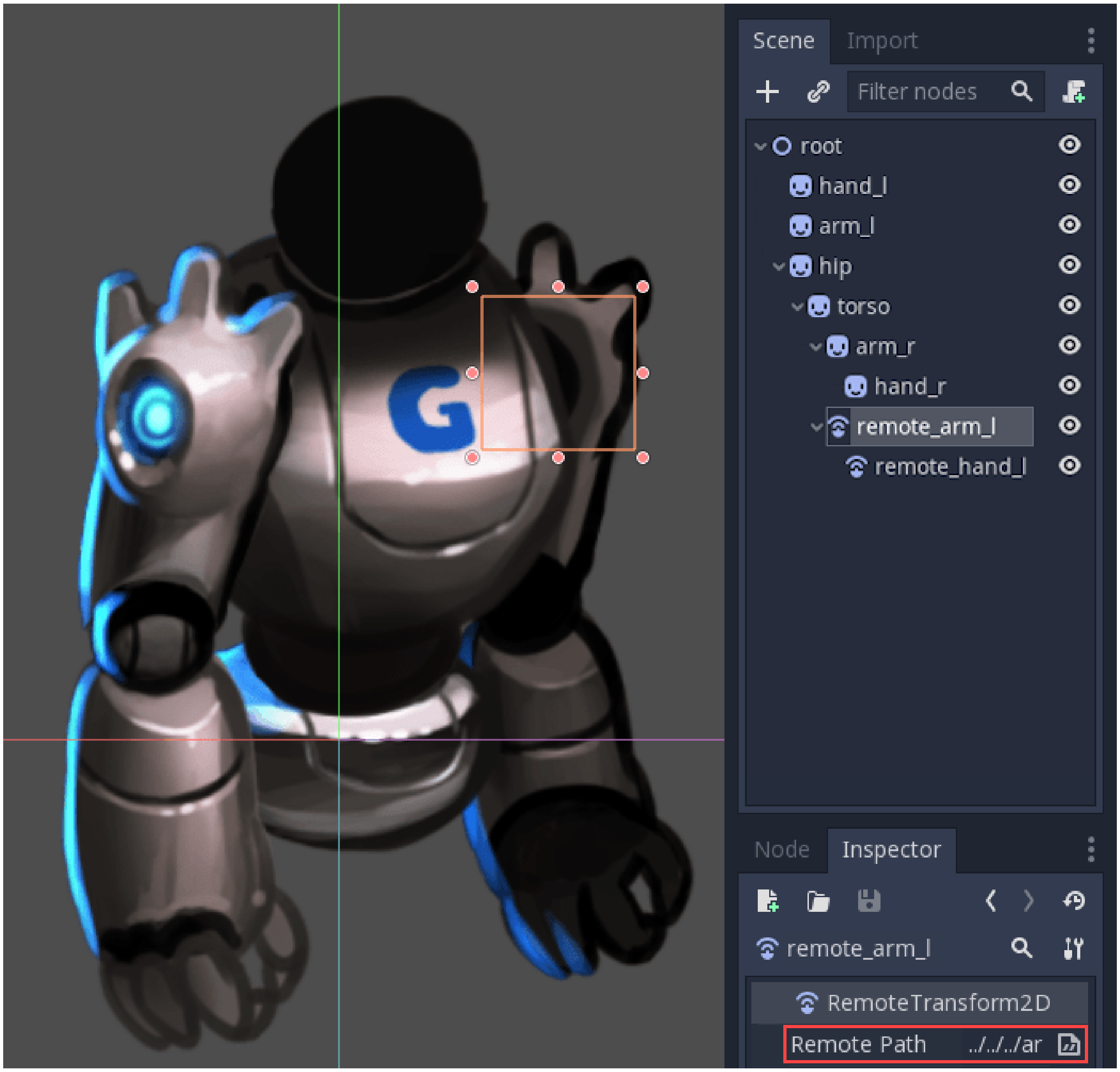
You can also fix depth ordering problems by adjusting the Z property of any node inheriting from Node2D.

RemoteTransform2D node

The RemoteTransform2D node transforms nodes somewhere else in the hierarchy. This node applies its own transform (including any transformation it inherits from its parents) to the remote node it targets.

This allows us to correct the visibility order of our elements, independently of the locations of those parts in the cutout hierarchy.

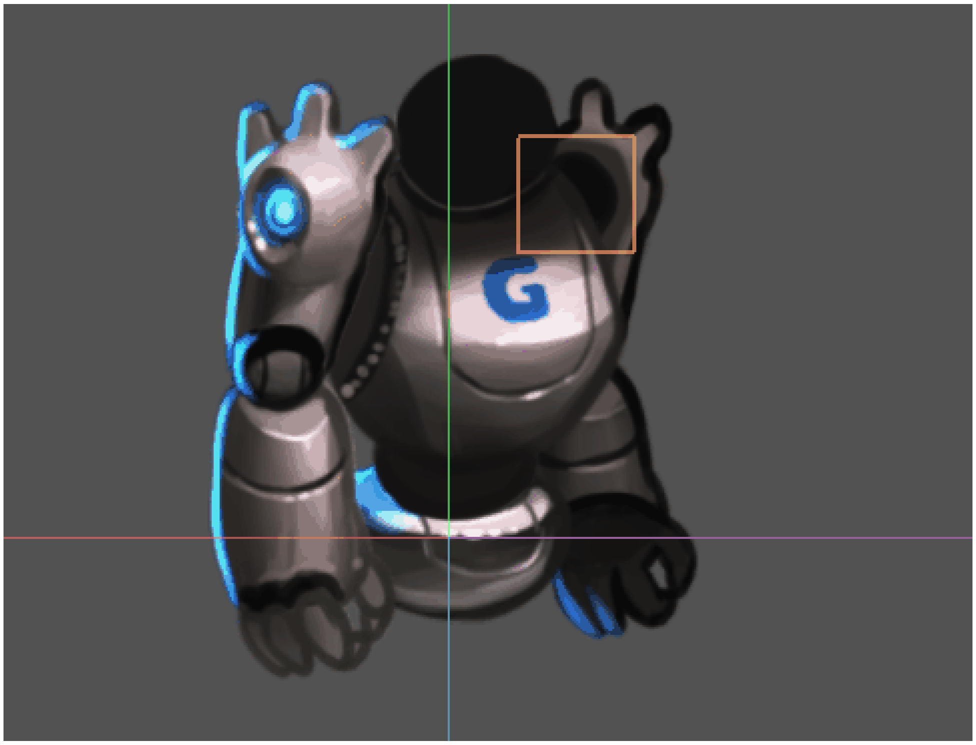
Create a `RemoteTransform2D` node as a child of the torso. Call it `remote_arm_l`. Create another RemoteTransform2D node inside the first and call it `remote_hand_l`. Use the `Remote Path` property of the two new nodes to target the `arm_l` and `hand_l` sprites respectively:



Scene
Import
Filter nodes
root
hand_l
arm_l
hip
torso
arm_r
hand_r
remote_arm_l
remote_hand_l
Node
Inspector
remote_arm_l
RemoteTransform2D

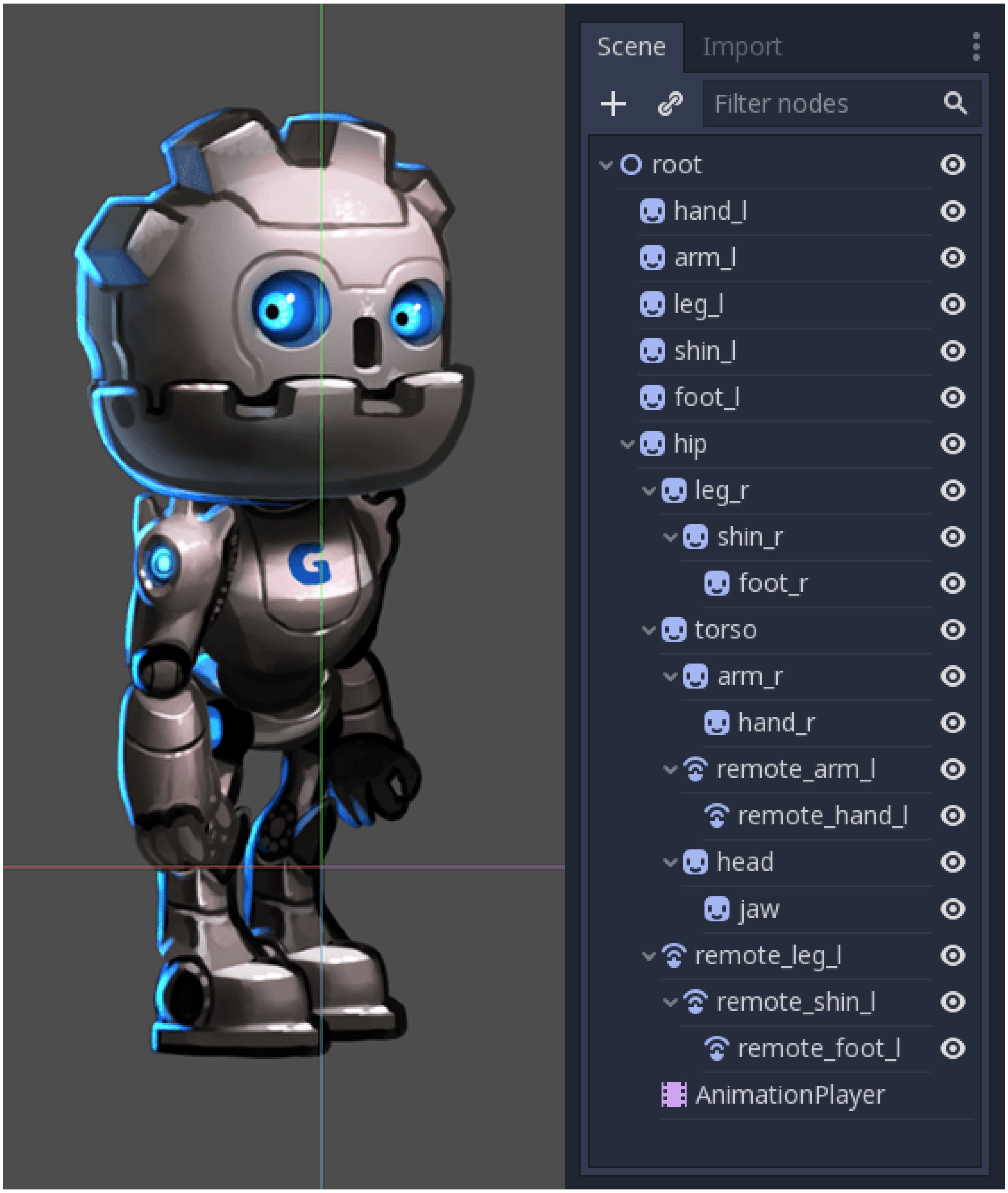
Remote Path ../../../ar

Moving the `RemoteTransform2D` nodes now moves the sprites. So we can create animations by adjusting the `RemoteTransform2D` transforms:



Completing the skeleton

Complete the skeleton by following the same steps for the rest of the parts. The resulting scene should look similar to this:


Scene
Import
Filter nodes
root
hand_l
arm_l
leg_l
shin_l
foot_l
hip
leg_r
shin_r
foot_r
torso
arm_r
hand_r
remote_arm_l
remote_hand_l
head
jaw
remote_leg_l
remote_shin_l
remote_foot_l
AnimationPlayer

The resulting rig will be easy to animate. By selecting the nodes and rotating them you can animate forward kinematics (FK) efficiently.

For simple objects and rigs this is fine, but there are limitations:

- Selecting sprites in the main viewport can become difficult in complex rigs. The scene tree ends up being used to select parts

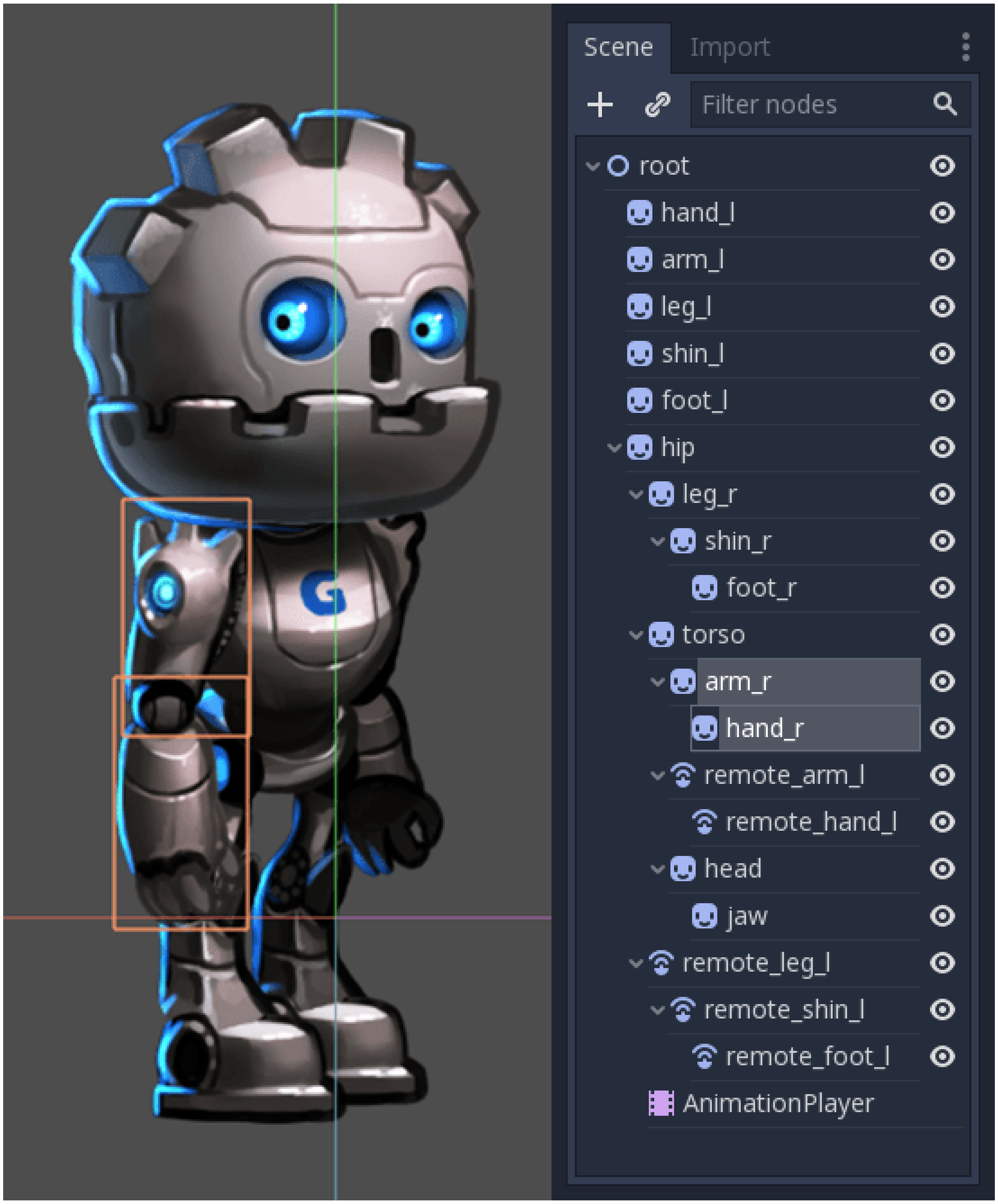
instead, which can be slower.
- Inverse Kinematics (IK) is useful for animating extremities like hands and feet, and can't be used with our rig in its current state.

To solve these problems we'll use Godot's skeletons.

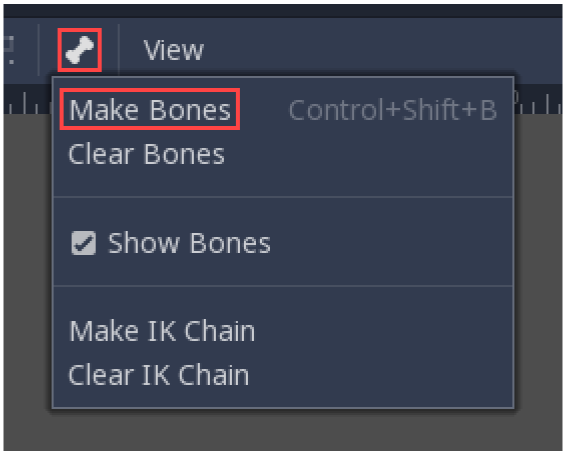
Skeletons

In Godot there is a helper to create "bones" between nodes. The bone-linked nodes are called skeletons.

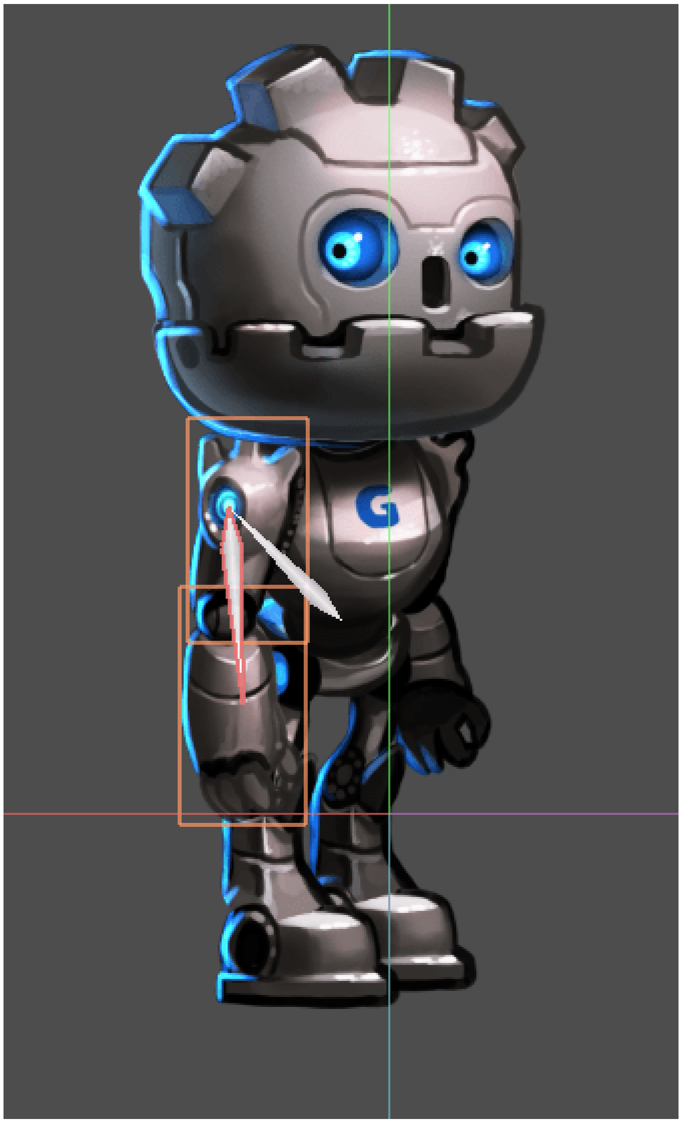
As an example, let's turn the right arm into a skeleton. To create a skeleton, a chain of nodes must be selected from top to bottom:


Scene
Import
Filter nodes
root
hand_l
arm_l
leg_l
shin_l
foot_l
hip
leg_r
shin_r
foot_r
torso
arm_r
hand_r
remote_arm_l
remote_hand_l
head
jaw
remote_leg_l
remote_shin_l
remote_foot_l
AnimationPlayer

Then, click on the Skeleton menu and select `Make Bones`.

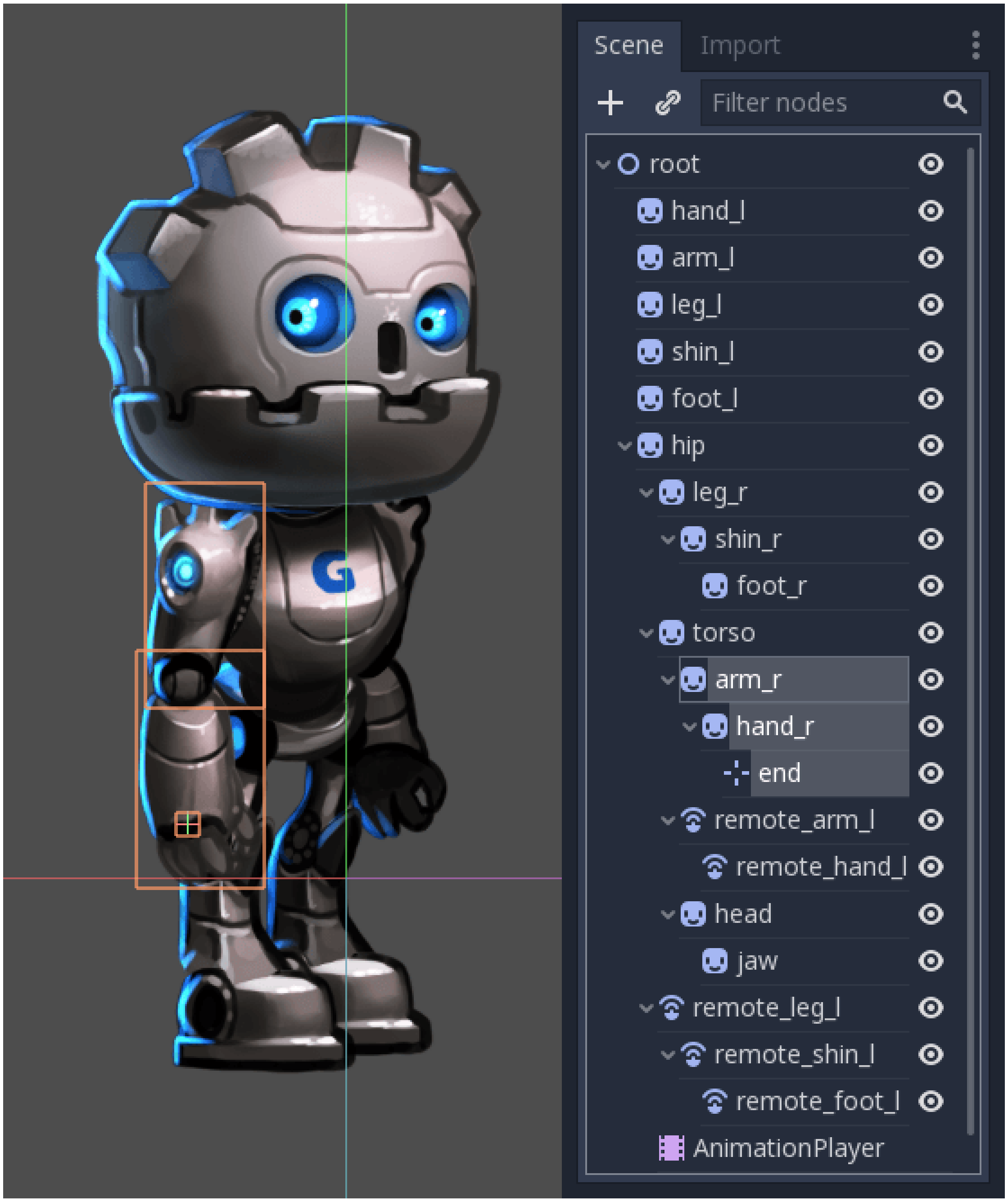

View
Make Bones Control+Shift+B
Clear Bones
☑ Show Bones
Make IK Chain
Clear IK Chain

This will add bones covering the arm, but the result may be surprising.

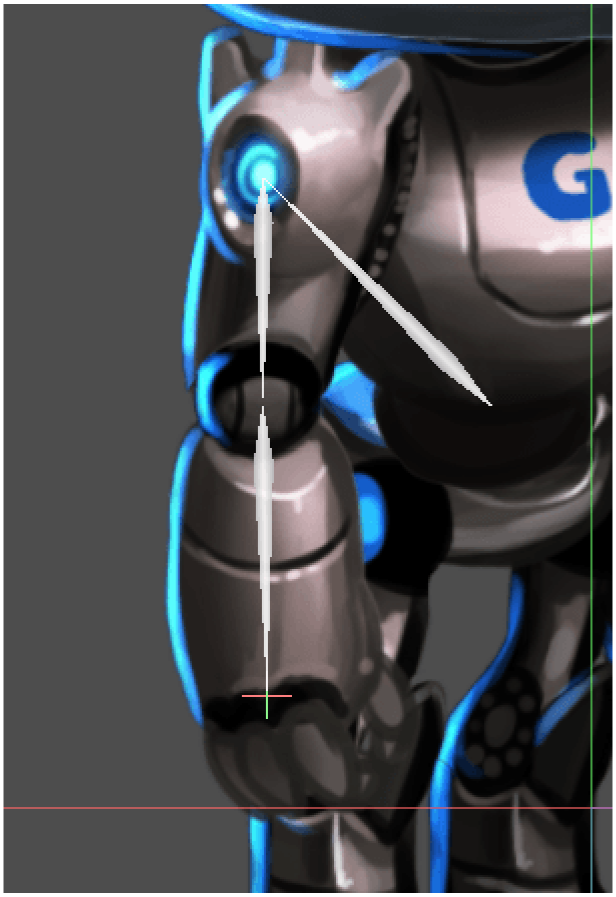


Why does the hand lack a bone? In Godot, a bone connects a node with its parent. And there's currently no child of the hand node. With this knowledge let's try again.

The first step is creating an endpoint node. Any kind of node will do, but Marker2D is preferred because it's visible in the editor. The endpoint node will ensure that the last bone has orientation.


Scene
Import
Filter nodes
root
hand_l
arm_l
leg_l
shin_l
foot_l
hip
leg_r
shin_r
foot_r
torso
arm_r
hand_r
end
remote_arm_l
remote_hand_l
head
jaw
remote_leg_l
remote_shin_l
remote_foot_l
AnimationPlayer

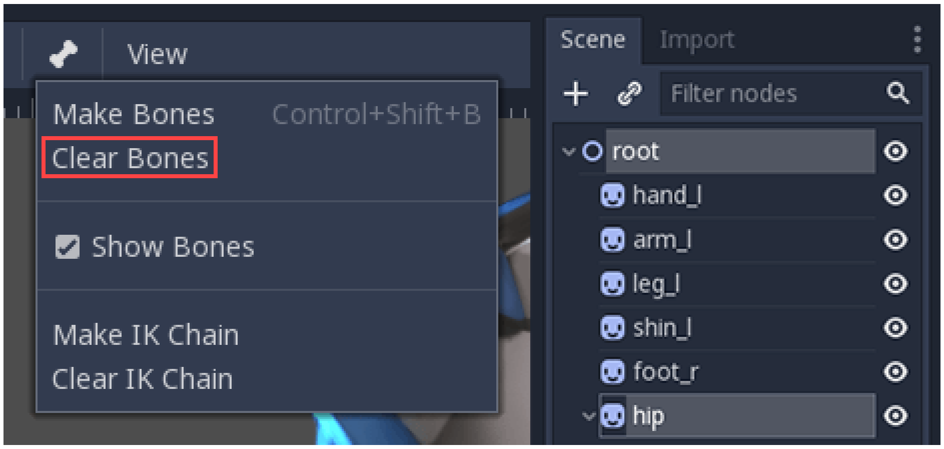
Now select the whole chain, from the endpoint to the arm and create bones:



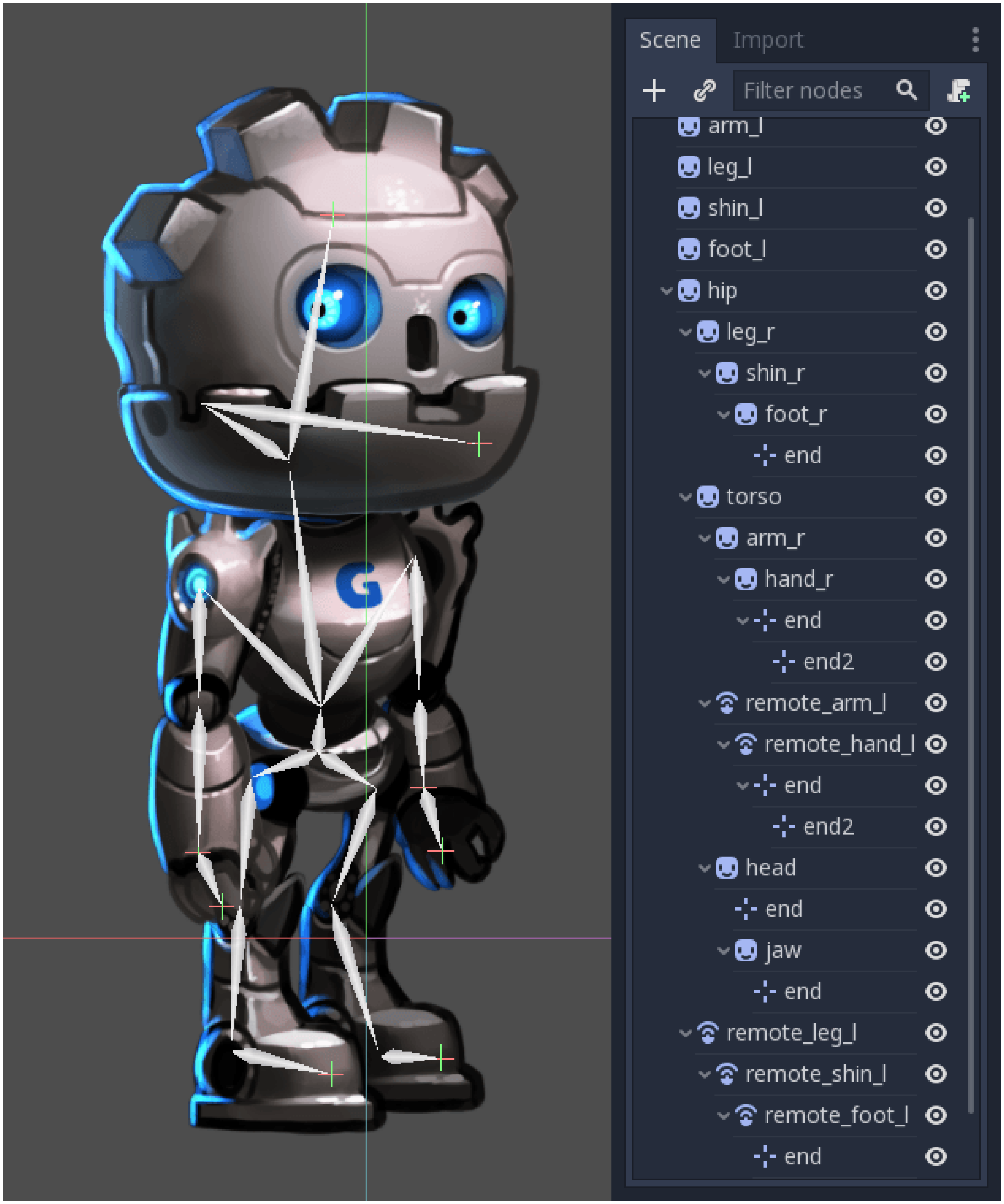
The result resembles a skeleton a lot more, and now the arm and forearm can be selected and animated.

Create endpoints for all important extremities. Generate bones for all articulable parts of the cutout, with the hip as the ultimate connection between all of them.

You may notice that an extra bone is created when connecting the hip and torso. Godot has connected the hip node to the scene root with a bone, and we don't want that. To fix this, select the root and hip node, open the Skeleton menu, click `clear bones`.


View
Make Bones Control+Shift+B
Clear Bones
☑ Show Bones
Make IK Chain
Clear IK Chain
Scene
Import
Filter nodes
root
hand_l
arm_l
leg_l
shin_l
foot_r
hip

Your final skeleton should look something like this:


Scene
Import
Filter nodes
arm_l
leg_l
shin_l
foot_l
hip
leg_r
shin_r
foot_r
end
torso
arm_r
hand_r
end
end2
remote_arm_l
remote_hand_l
end
end2
head
end
jaw
end
remote_leg_l
remote_shin_l
remote_foot_l
end

You might have noticed a second set of endpoints in the hands. This will make sense soon.

Now that the whole figure is rigged, the next step is setting up the IK chains. IK chains allow for more natural control of extremities.

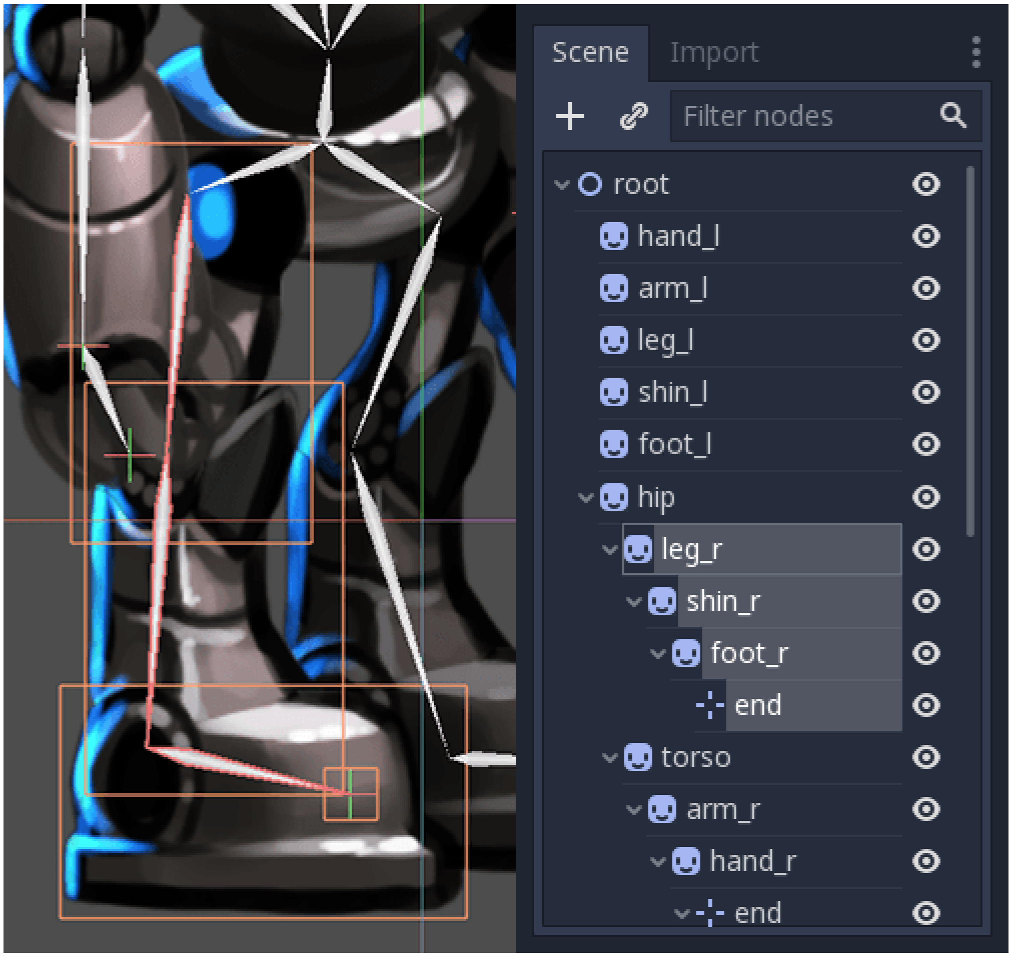
IK chains

IK stands for inverse kinematics. It's a convenient technique for animating the position of hands, feet and other extremities of rigs like the one we've made. Imagine you want to pose a character's foot in a specific position on the ground. Without IK chains, each motion of the foot would require rotating and positioning several other bones (the shin and the thigh at least). This would be quite complex and lead to imprecise results. IK allows us to move the foot directly while the shin and thigh self-adjust.

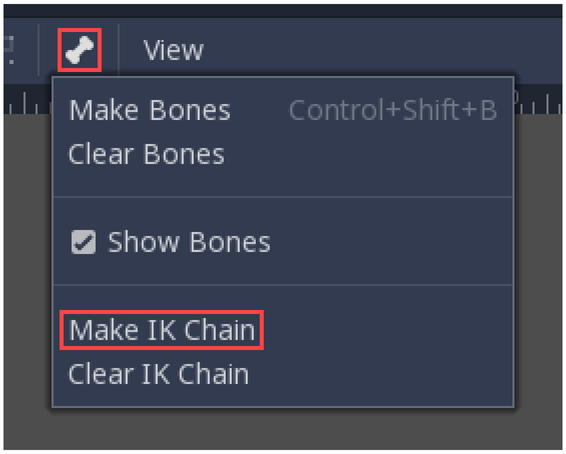
Note

IK chains in Godot currently work in the editor only, not at runtime. They are intended to ease the process of setting keyframes, and are not currently useful for techniques like procedural animation.

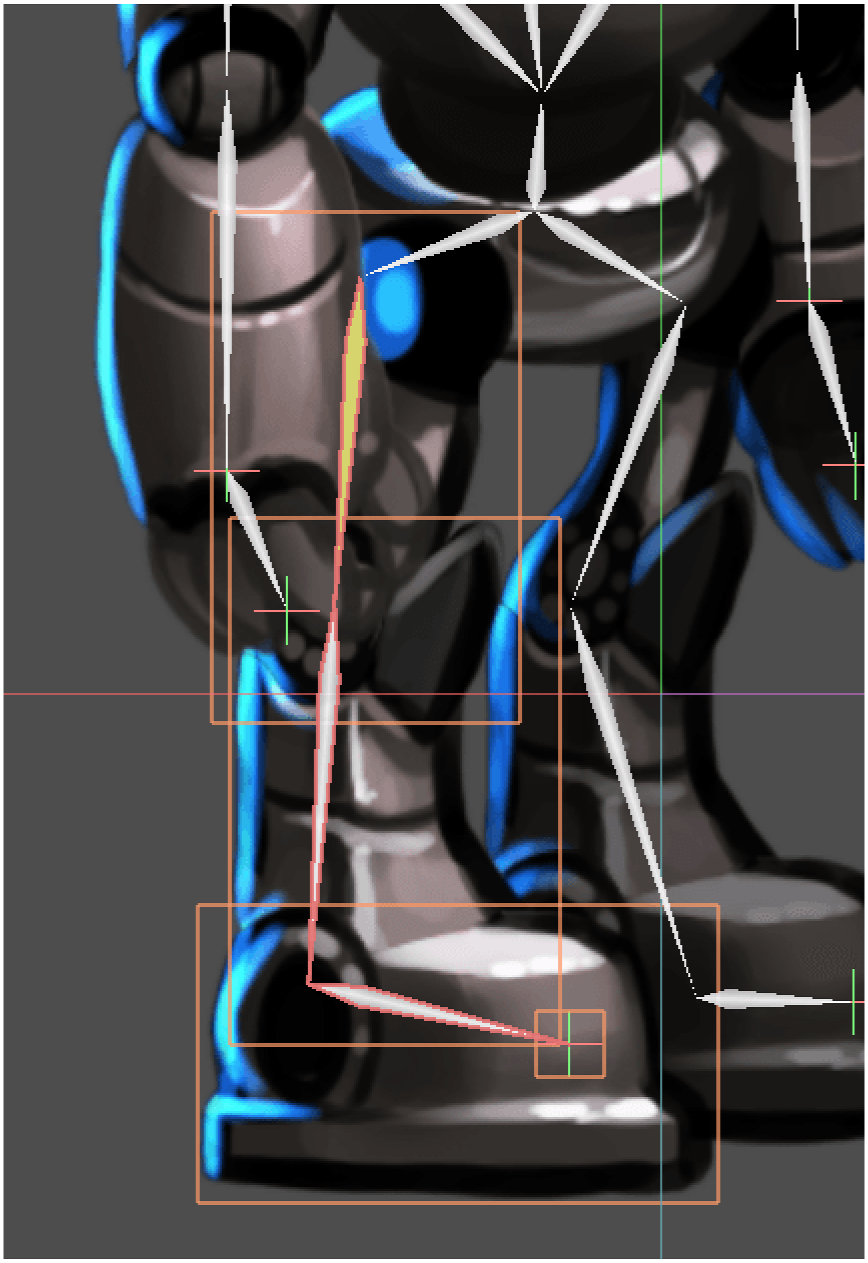
To create an IK chain, select a chain of bones from endpoint to the base for the chain. For example, to create an IK chain for the right leg, select the following:


Scene
Import
Filter nodes
root
hand_l
arm_l
leg_l
shin_l
foot_l
hip
leg_r
shin_r
foot_r
end
torso
arm_r
hand_r
end

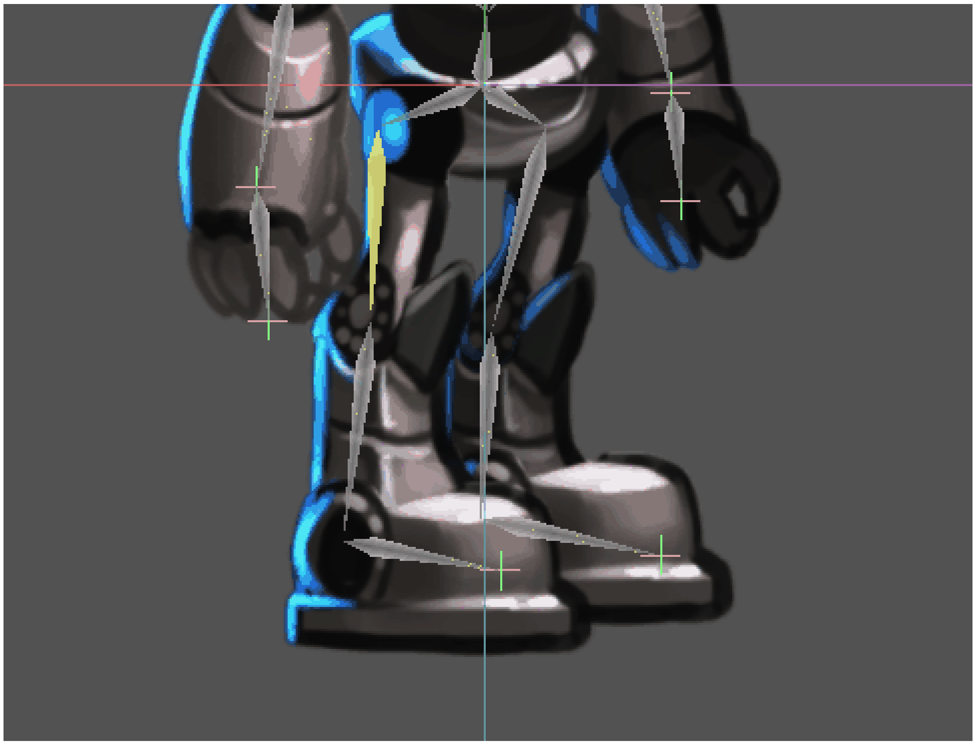
Then enable this chain for IK. Go to Edit > Make IK Chain.


View
Make Bones Control+Shift+B
Clear Bones
☑ Show Bones
Make IK Chain
Clear IK Chain

As a result, the base of the chain will turn *Yellow*.



Once the IK chain is set up, grab any child or grand-child of the base of the chain (e.g. a foot), and move it. You'll see the rest of the chain adjust as you adjust its position.

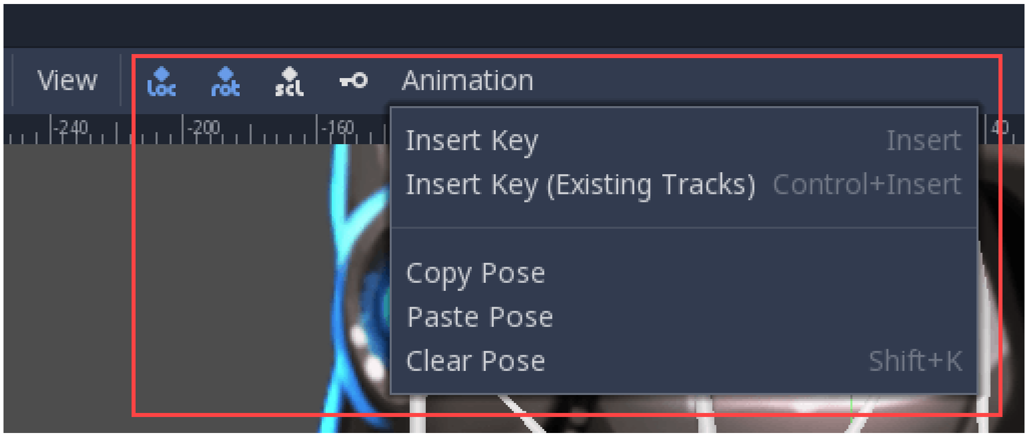


Animation tips

The following section will be a collection of tips for creating animation for your cutout rigs. For more information on how the animation system in Godot works, see Introduction to the animation features.

Setting keyframes and excluding properties

Special contextual elements appear in the top toolbar when the animation editor window is open:


View
loc
rot
scl
Animation
Insert Key
Insert
Insert Key (Existing Tracks)
Control+Insert
Copy Pose
Paste Pose
Clear Pose
Shift+K
-240
-200
-160
40

The key button inserts location, rotation, and scale keyframes for the selected objects or bones at the current playhead position.

The "loc", "rot", and "scl" toggle buttons to the left of the key button modify its function, allowing you to specify which of the three properties keyframes will be created for.

Here's an illustration of how this can be useful: Imagine you have a node which already has two keyframes animating its scale only. You want to add an overlapping rotation movement to the same node. The rotation movement should begin and end at different times from the scale change that's already set up. You can use the toggle buttons to have only rotation information added when you add a new keyframe. This way, you can avoid adding unwanted scale keyframes which would disrupt the existing scale animation.

Creating a rest pose

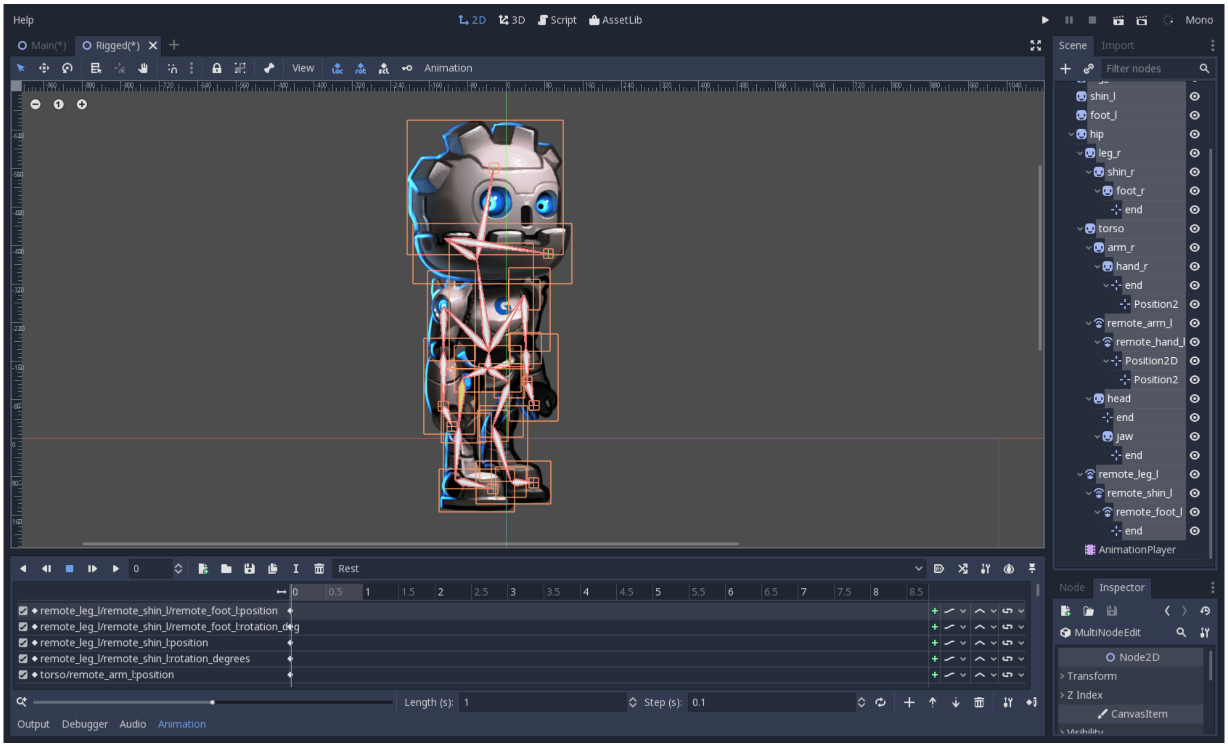
Think of a rest pose as a default pose that your cutout rig should be set to when no other pose is active in your game. Create a rest pose as follows:

1. Make sure the rig parts are positioned in what looks like a "resting" arrangement.

2. Create a new animation, rename it "rest".
3. Select all nodes in your rig (box selection should work fine).

4. Make sure the "loc", "rot", and "scl" toggle buttons are all active in the toolbar.

5. Press the key button. Keys will be inserted for all selected parts storing their current arrangement. This pose can now be recalled when necessary in your game by playing the "rest" animation you've created.


Help
2D
3D
Script
AssetLib
Mono
Main(*)
Rigged(*)
View
Animation
Scene
Import
Filter nodes
shin_l
foot_l
hip
leg_r
shin_r
foot_r
end
torso
arm_r
hand_r
end
Position2
remote_arm_l
remote_hand_l
Position2D
Position2
head
end
jaw
end
remote_leg_l
remote_shin_l
remote_foot_l
end
AnimationPlayer
Node
Inspector
MultiNodeEdit
Node2D
Transform
Z Index
CanvasItem
Visibility
Output
Debugger
Audio
Animation

Rest

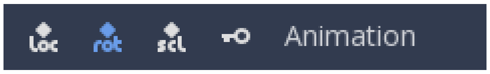
- remote_leg_l/remote_shin_l/remote_foot_l:position
- remote_leg_l/remote_shin_l/remote_foot_l:rotation_deg
- remote_leg_l/remote_shin_l:position
- remote_leg_l/remote_shin_l:rotation_degrees
- torso/remote_arm_l:position

Length (s): 1
Step (s): 0.1

Modifying rotation only

When animating a cutout rig, often it's only the rotation of the nodes that needs to change. Location and scale are rarely used.

So when inserting keys, you might find it convenient to have only the "rot" toggle active most of the time:



This will avoid the creation of unwanted animation tracks for position and scale.

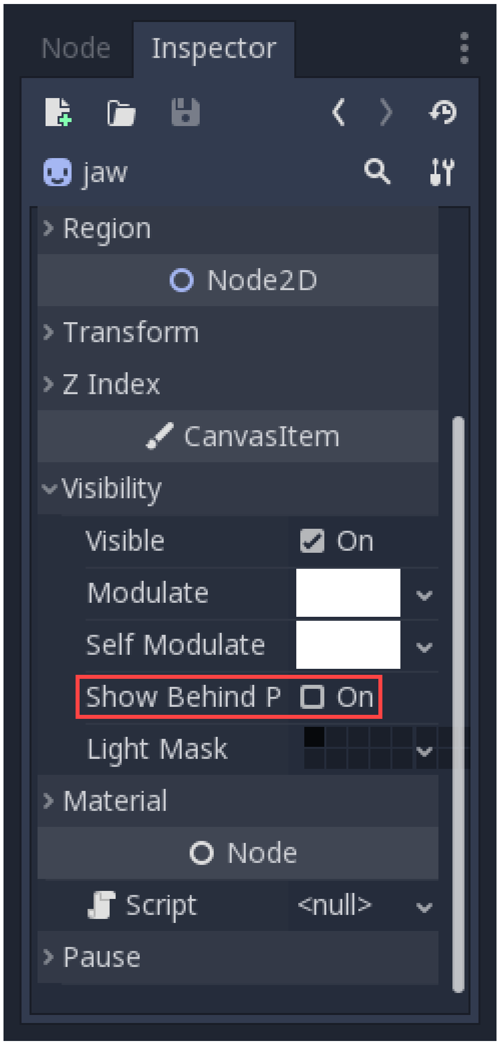
Keyframing IK chains

When editing IK chains, it's not necessary to select the whole chain to add keyframes. Selecting the endpoint of the chain and inserting a keyframe will automatically insert keyframes for all other parts of the chain too.

Visually move a sprite behind its parent

Sometimes it is necessary to have a node change its visual depth relative to its parent node during an animation. Think of a character facing the camera, who pulls something out from behind his back and holds it out in front of him. During this animation the whole arm and the object in his hand would need to change their visual depth relative to the body of the character.

To help with this there's a keyframable "Behind Parent" property on all Node2D-inheriting nodes. When planning your rig, think about the movements it will need to perform and give some thought to how you'll use "Behind Parent" and/or RemoteTransform2D nodes. They provide overlapping functionality.


Node
Inspector
jaw
Region
Node2D
Transform
Z Index
CanvasItem
Visibility

Visible ☑ On

Modulate

Self Modulate

Show Behind P ☐ On

Light Mask

Material
Node

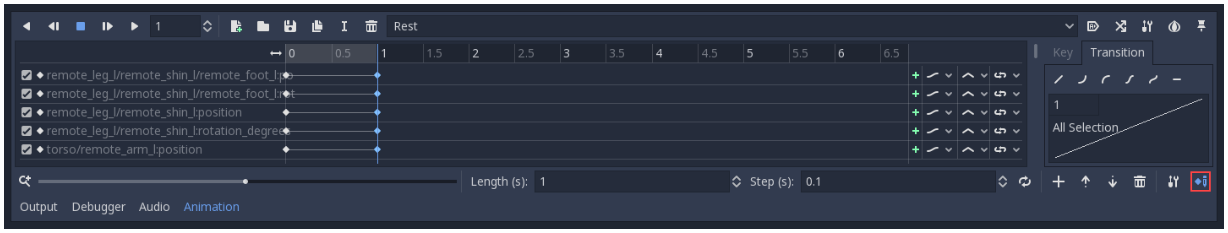
Script <null>

Pause

Setting easing curves for multiple keys

To apply the same easing curve to multiple keyframes at once:

1. Select the relevant keys.
2. Click on the pencil icon in the bottom right of the animation panel. This will open the transition editor.
3. In the transition editor, click on the desired curve to apply it.


Rest
remote_leg_l/remote_shin_l/remote_foot_l:p
remote_leg_l/remote_shin_l/remote_foot_l:r
remote_leg_l/remote_shin_l:position
remote_leg_l/remote_shin_l:rotation_degrees
torso/remote_arm_l:position
Key
Transition
All Selection
Output
Debugger
Audio
Animation

Length (s): 1

Step (s): 0.1

2D Skeletal deform

Skeletal deform can be used to augment a cutout rig, allowing single pieces to deform organically (e.g. antennae that wobble as an insect character walks).

This process is described in a separate tutorial.

2D skeletons

Introduction

When working with 3D, skeletal deforms are common for characters and creatures and most 3D modeling applications support it. For 2D, as this function is not used as often, it's difficult to find mainstream software aimed for this.

One option is to create animations in third-party software such as Spine or Dragonbones. From Godot 3.1 onwards, though, this functionality is supported built-in.

Why would you want to do skeletal animations directly in Godot? The answer is that there are many advantages to it:

- Better integration with the engine, so less hassle importing and editing from an external tool.
- Ability to control particle systems, shaders, sounds, call scripts, colors, transparency, etc. in animations.
- The built-in skeletal system in Godot is very efficient and designed for performance.

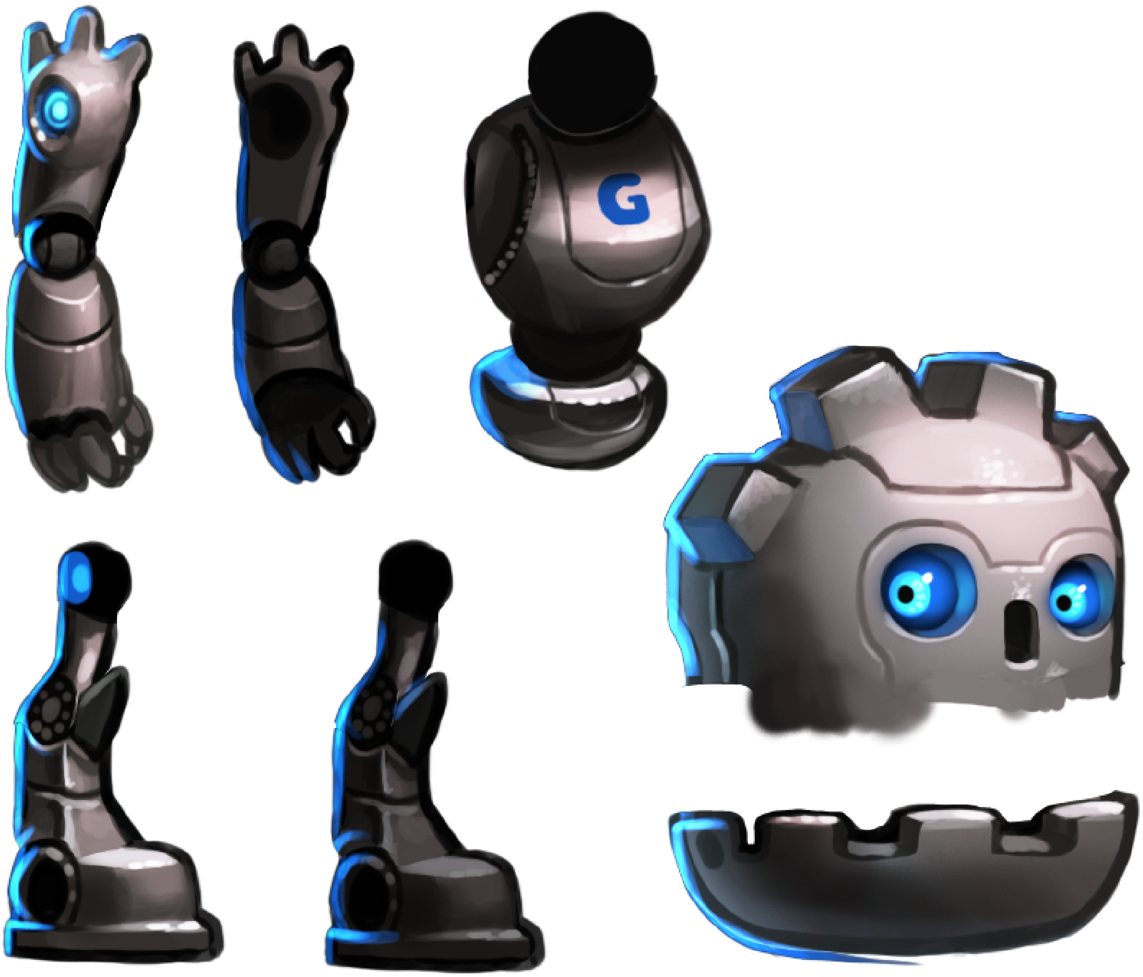
The following tutorial will, then, explain 2D skeletal deformations.

Setup

See also

Before starting, we recommend you to go through the Cutout animation tutorial to gain a general understanding of animating within Godot.

For this tutorial, we will be using a single image to construct our character. Download it from `gBot_pieces.png` or save the image below.



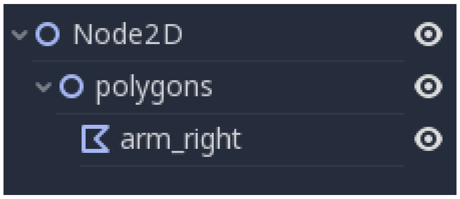
It is also advised to download the final character image `gBot_complete.png` to have a good reference for putting the different pieces together.

G

Creating the polygons

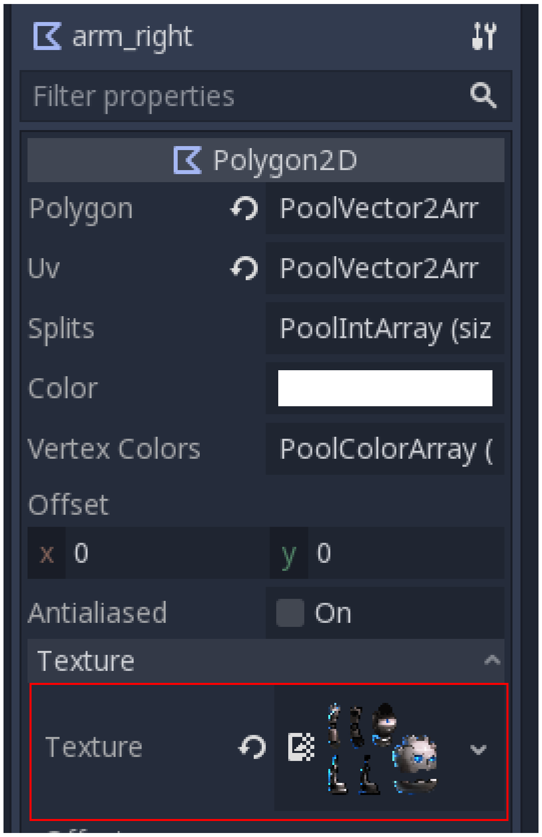
Create a new scene for your model (if it's going to be an animated character, you may want to use a `CharacterBody2D`). For ease of use, an empty 2D node is created as a root for the polygons.

Begin with a `Polygon2D` node. There is no need to place it anywhere in the scene for now, so simply create it like this:



- Node2D
  - polygons
    - arm_right

Select it and assign the texture with the character pieces you have downloaded before:



arm_right

Filter properties

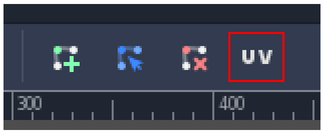
Polygon2D

Polygon	PoolVector2Arr
Uv	PoolVector2Arr
Splits	PoolIntArray (siz
Color	
Vertex Colors	PoolColorArray (
Offset	x 0 y 0
Antialiased	☐ On

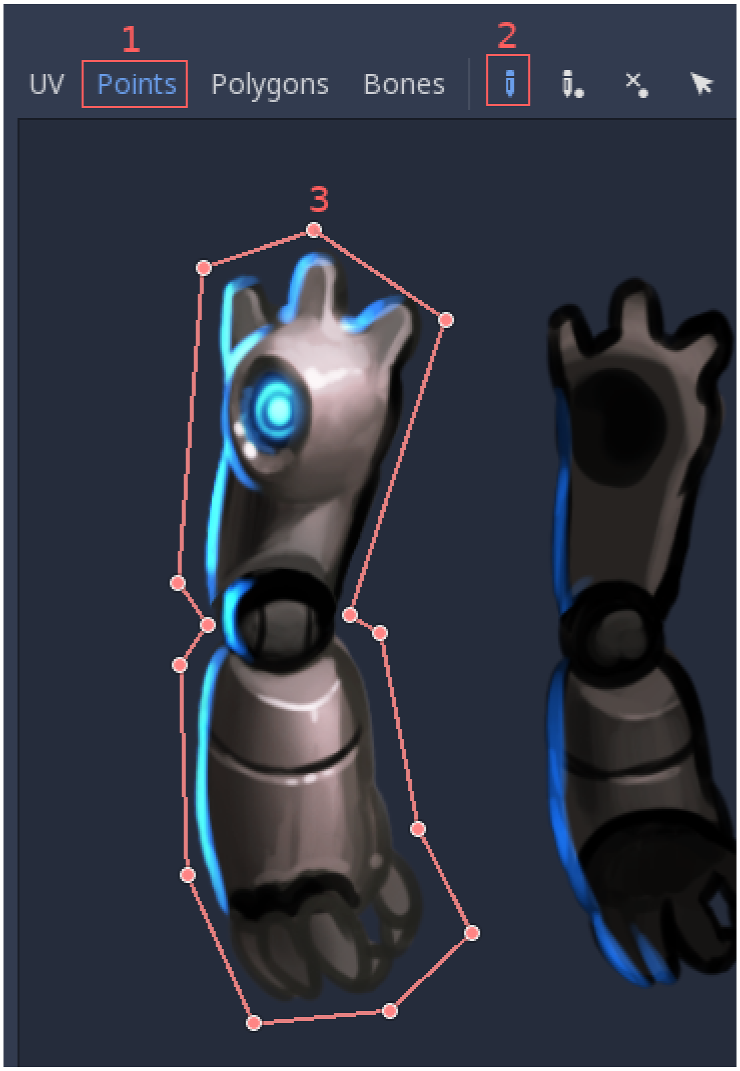
Texture

Texture	

Drawing a polygon directly is not advised. Instead, open the "UV" dialog for the polygon:



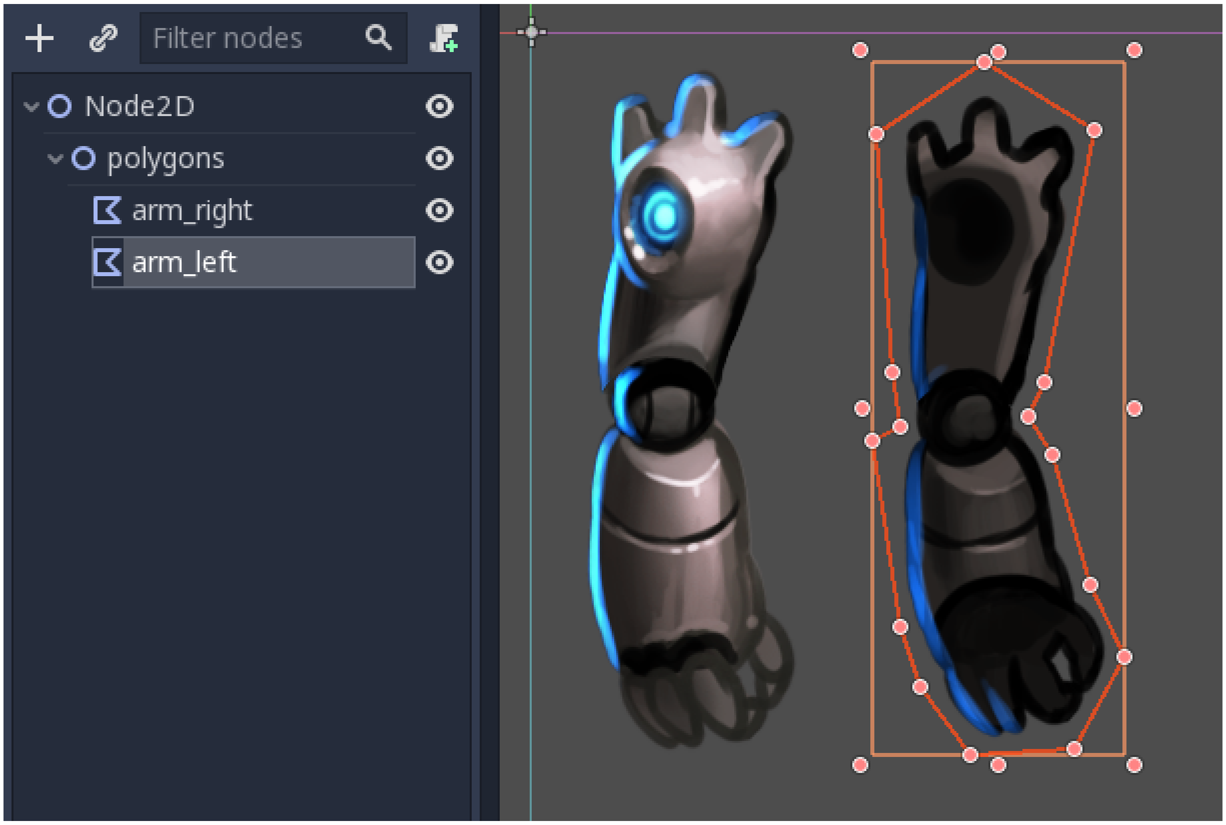
Head over to the *Points* mode, select the pencil and draw a polygon around the desired piece:


1
2
3
UV
Points
Polygons
Bones

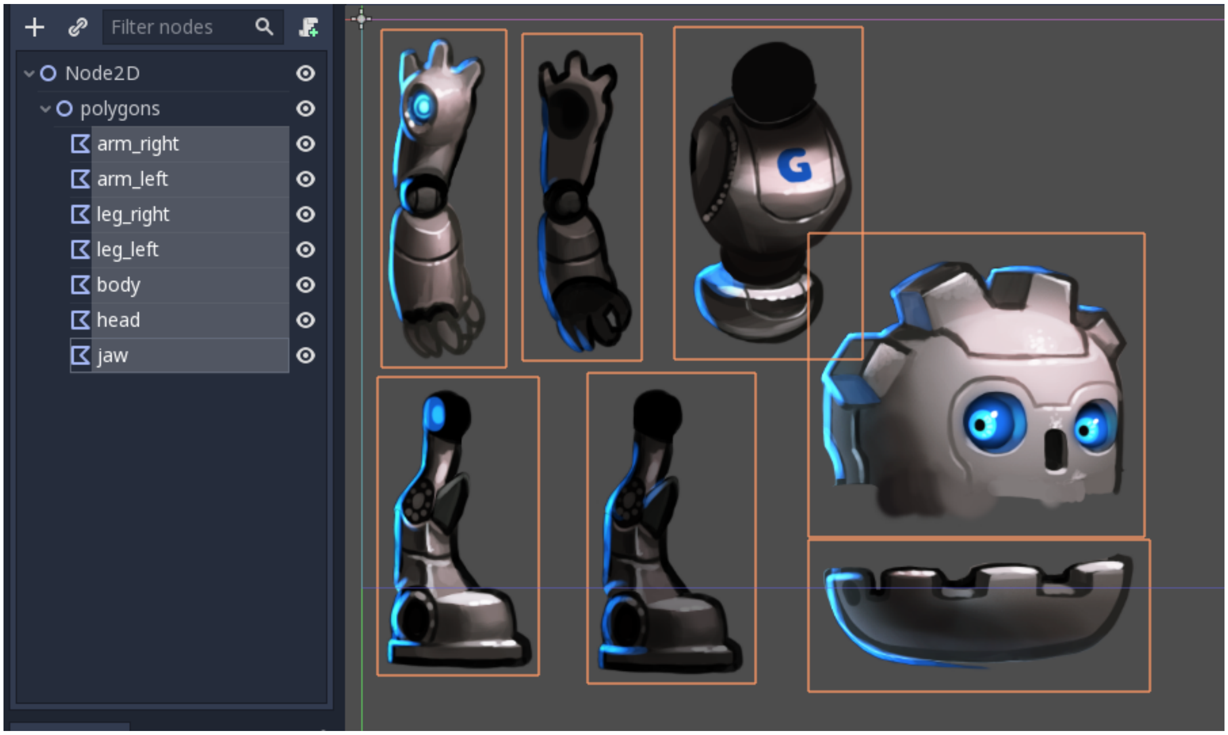
Duplicate the polygon node and give it a proper name. Then, enter the "UV" dialog again and replace the old polygon with another one in the new desired piece.

When you duplicate nodes and the next piece has a similar shape, you can edit the previous polygon instead of drawing a new one.

After moving the polygon, remember to update the UV by selecting **Edit > Copy Polygon to UV** in the Polygon 2D UV Editor.

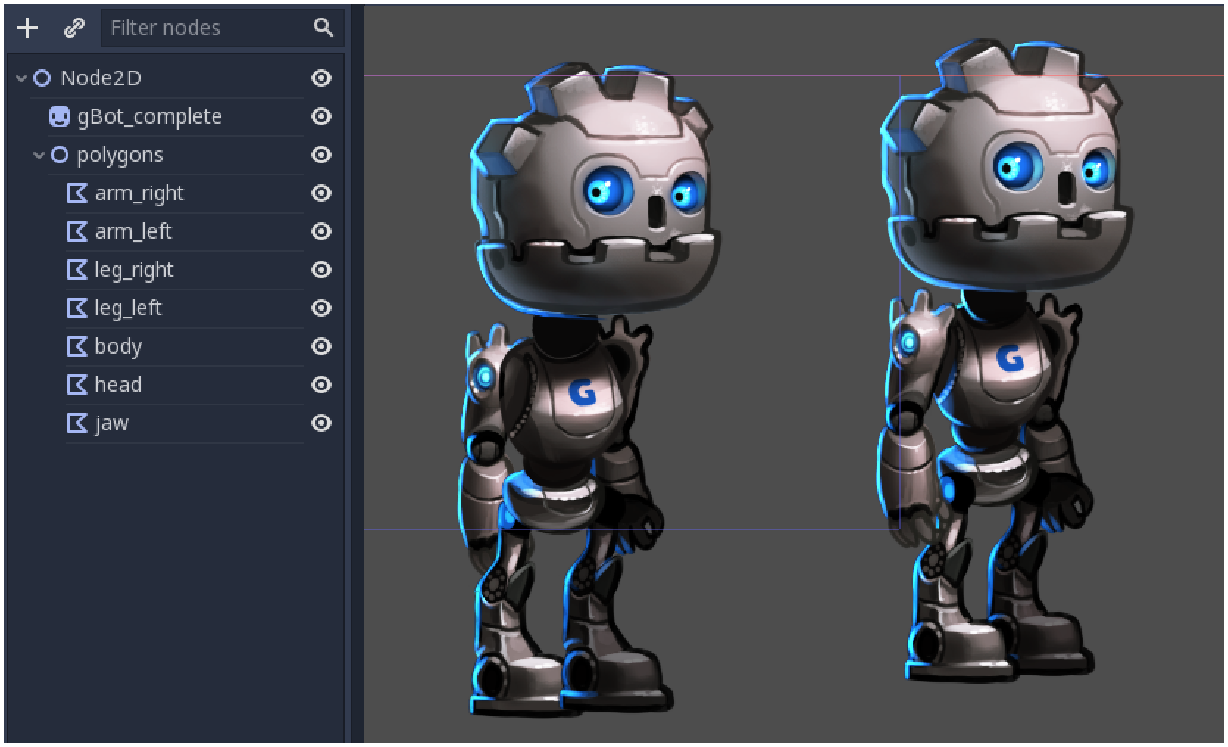

Filter nodes
Node2D
polygons
arm_right
arm_left

Keep doing this until you mapped all pieces.

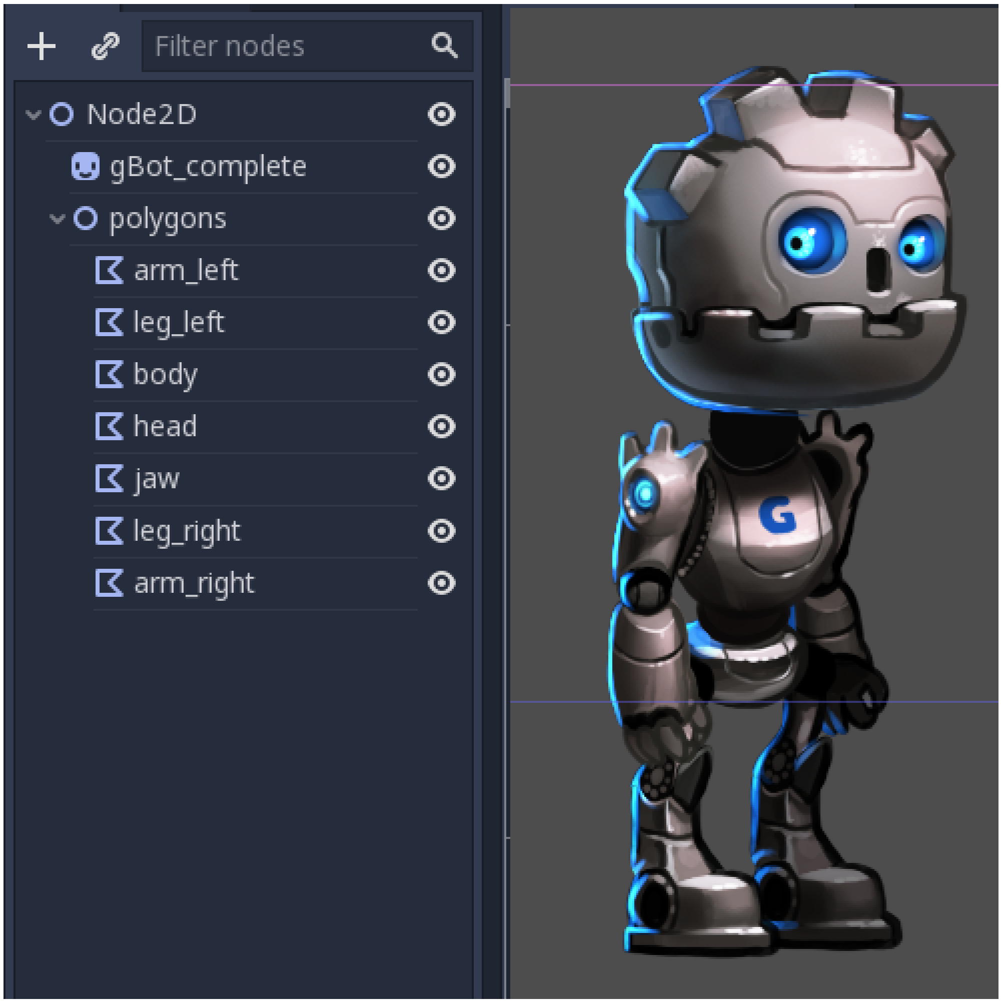

Filter nodes
Node2D
polygons
arm_right
arm_left
leg_right
leg_left
body
head
jaw

You will notice that pieces for nodes appear in the same layout as they do in the original texture. This is because by default, when you draw a polygon, the UV and points are the same.

Rearrange the pieces and build the character. This should be pretty quick. There is no need to change pivots, so don't bother making sure rotation pivots for each piece are right; you can leave them be for now.


Filter nodes
Node2D
gBot_complete
polygons
arm_right
arm_left
leg_right
leg_left
body
head
jaw

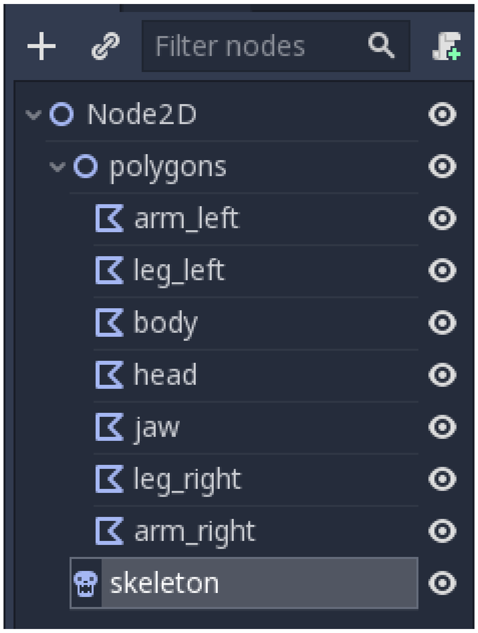
Ah, the visual order of the pieces is not correct yet, as some are covering wrong pieces. Rearrange the order of the nodes to fix this:


Filter nodes
Node2D
gBot_complete
polygons
arm_left
leg_left
body
head
jaw
leg_right
arm_right
G

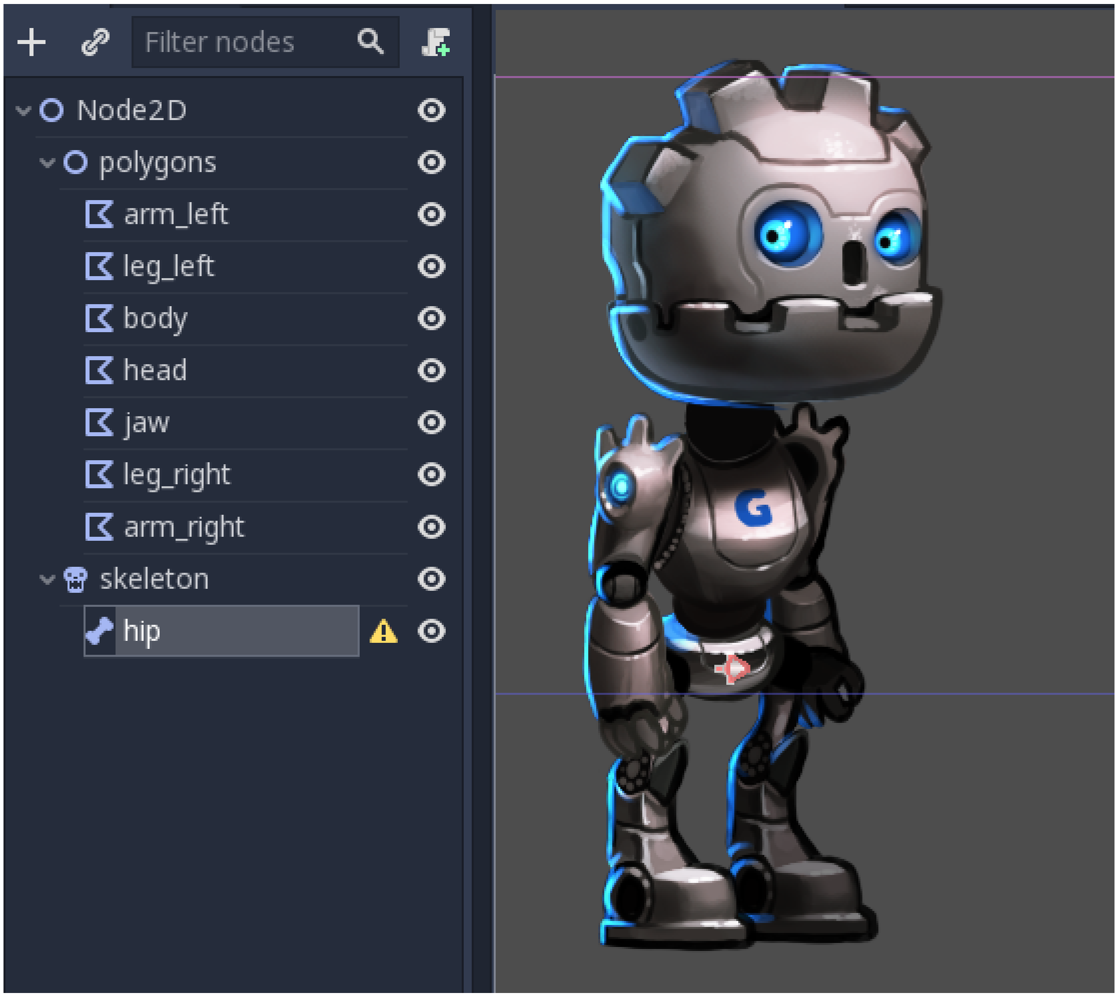
And there you go! It was definitely much easier than in the cutout tutorial.

Creating the skeleton

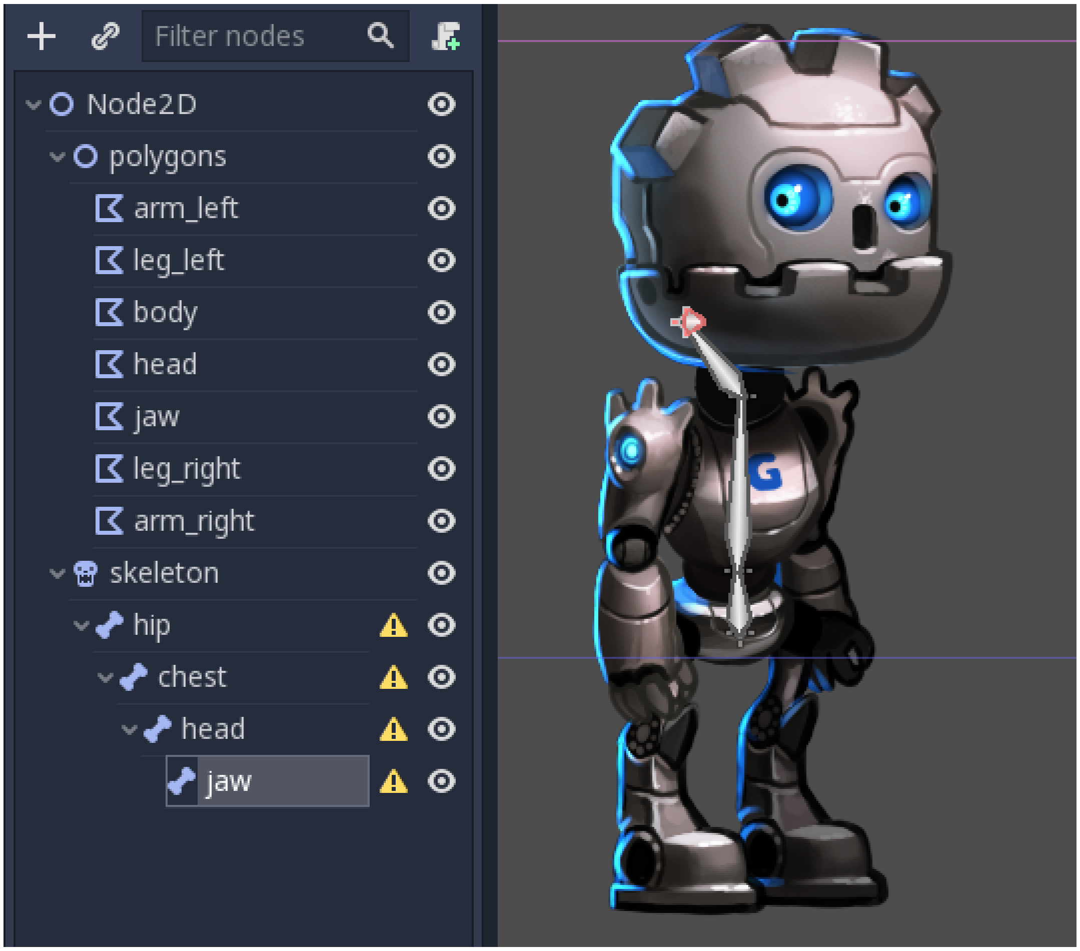
Create a `Skeleton2D` node as a child of the root node. This will be the base of our skeleton:


Filter nodes
Node2D
polygons
arm_left
leg_left
body
head
jaw
leg_right
arm_right
skeleton

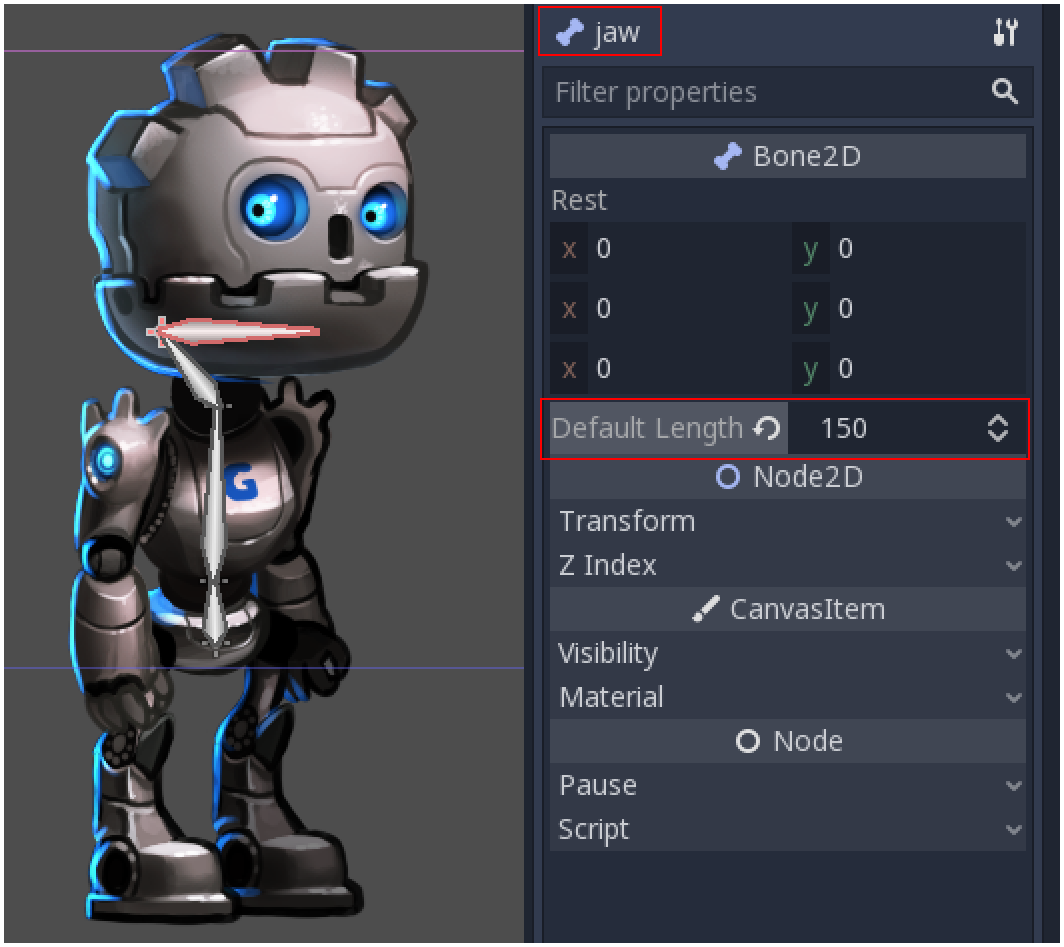
Create a `Bone2D` node as a child of the skeleton. Put it on the hip (usually skeletons start here). The bone will be pointing to the right, but you can ignore this for now.


Filter nodes
Node2D
polygons
arm_left
leg_left
body
head
jaw
leg_right
arm_right
skeleton
hip

Keep creating bones in hierarchy and naming them accordingly.

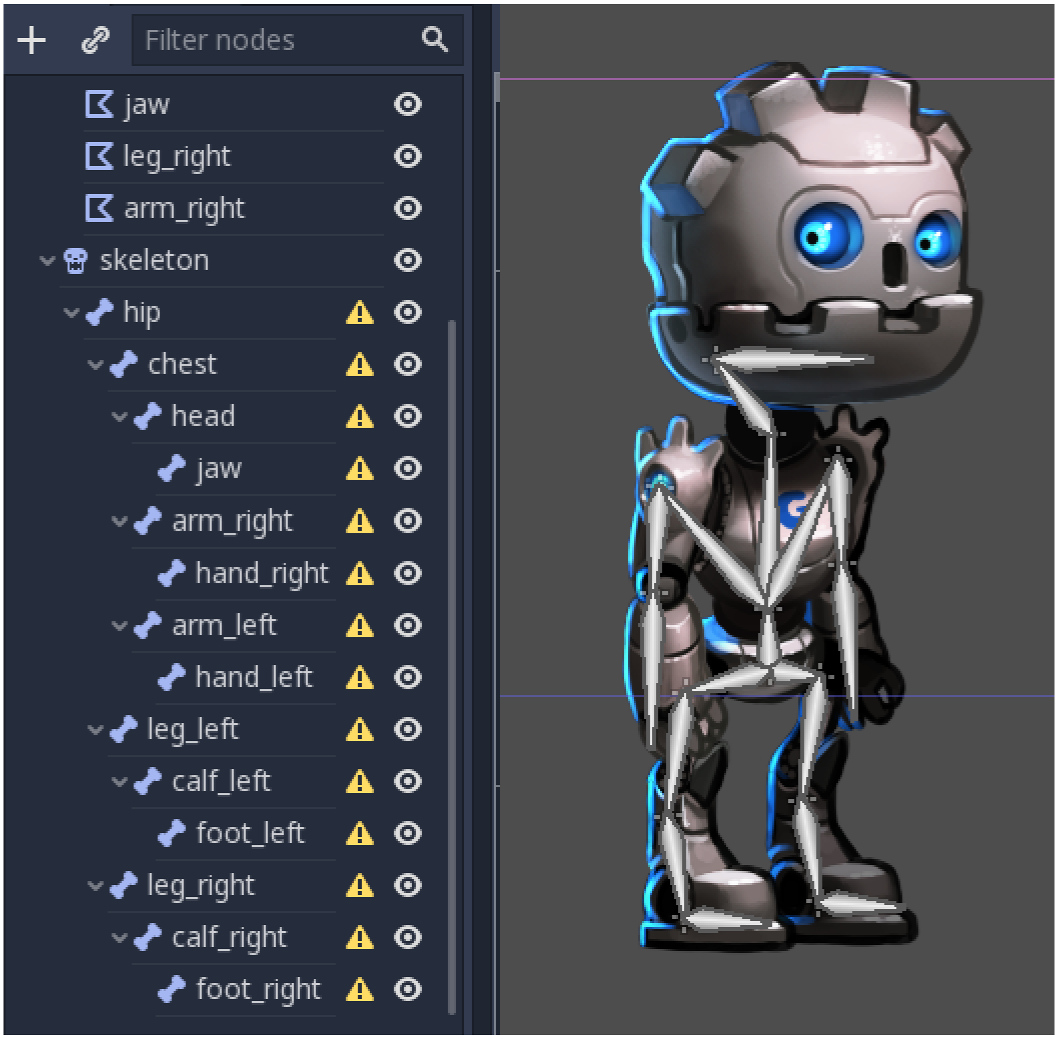

Filter nodes
Node2D
polygons
arm_left
leg_left
body
head
jaw
leg_right
arm_right
skeleton
hip
chest
head
jaw

At the end of this chain, there will be a *jaw* node. It is, again, very short and pointing to the right. This is normal for bones without children. The length of *tip* bones can be changed with a property in the inspector:

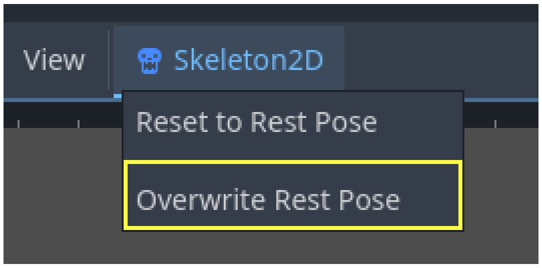

jaw
Filter properties
Bone2D
Rest
x 0 y 0
x 0 y 0
x 0 y 0
Default Length 150
Node2D
Transform
Z Index
CanvasItem
Visibility
Material
Node
Pause
Script

In this case, we don't need to rotate the bone (coincidentally the jaw points right in the sprite), but in case you need to, feel free to do it. Again, this is only really needed for tip bones as nodes with children don't usually need a length or a specific rotation.

Keep going and build the whole skeleton:


Filter nodes
jaw
leg_right
arm_right
skeleton
hip
chest
head
jaw
arm_right
hand_right
arm_left
hand_left
leg_left
calf_left
foot_left
leg_right
calf_right
foot_right

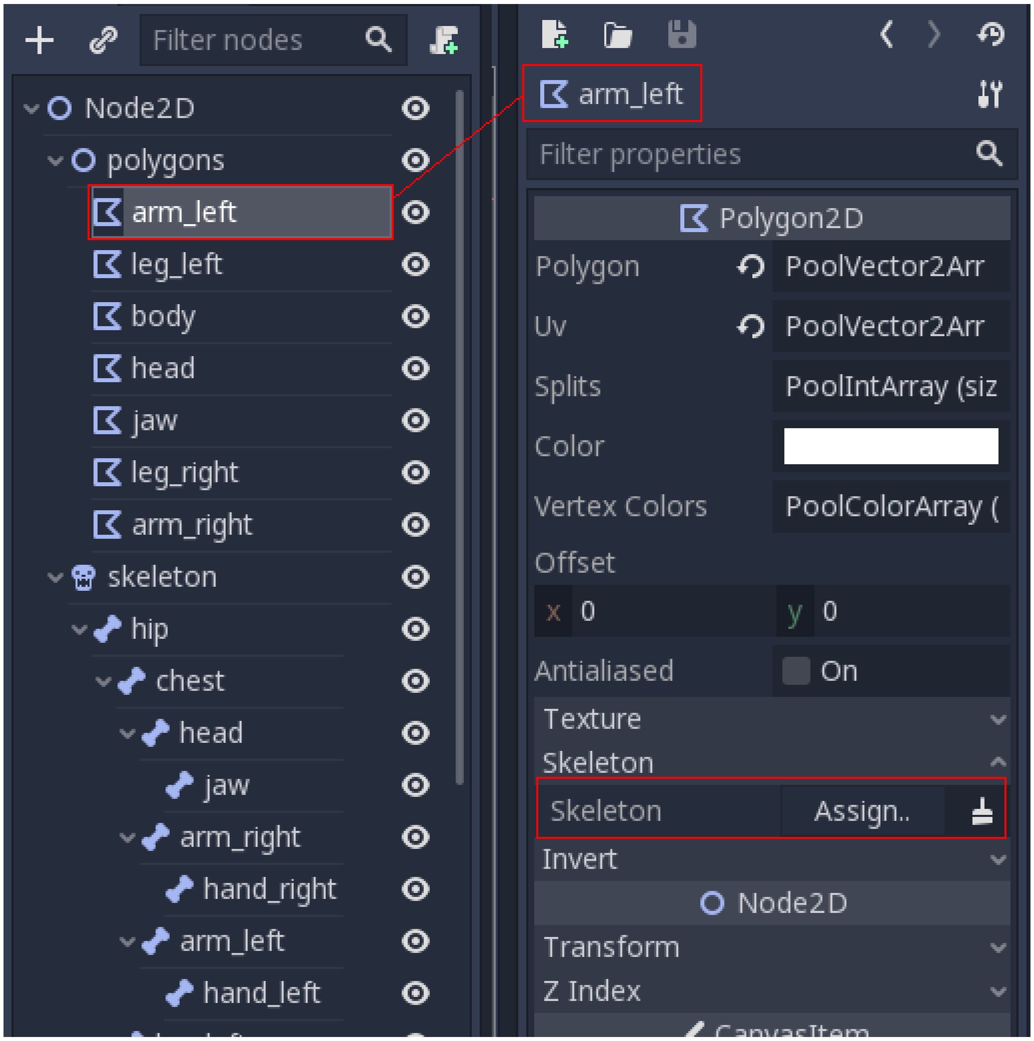
You will notice that all bones raise a warning about a missing rest pose. A rest pose is the default pose for a skeleton, you can come back to it anytime you want (which is very handy for animating). To set one click on the *skeleton* node in the scene tree, then click on the `Skeleton2D` button in the toolbar, and select `Overwrite Rest Pose` from the dropdown menu.


View
Skeleton2D
Reset to Rest Pose
Overwrite Rest Pose

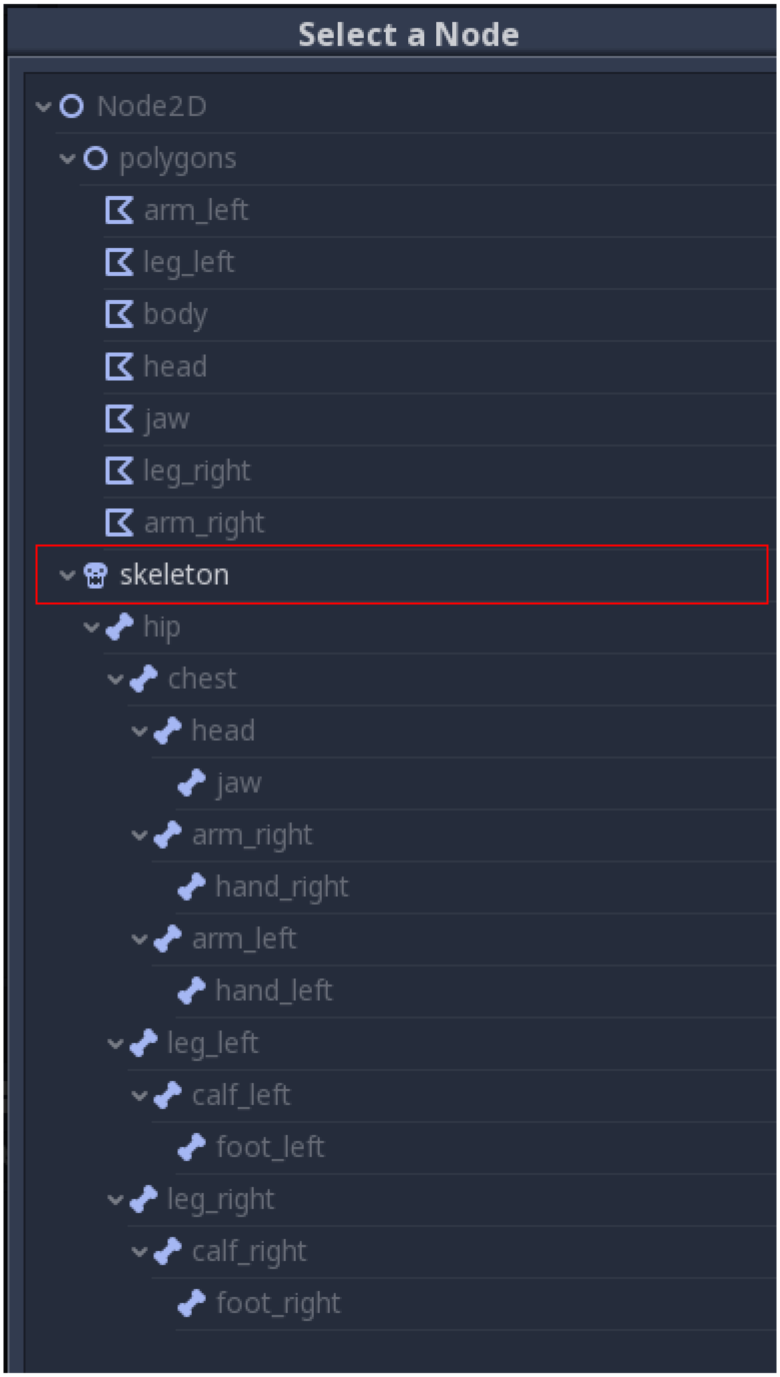
The warnings will go away. If you modify the skeleton (add/remove bones) you will need to set the rest pose again.

Deforming the polygons

Select the previously created polygons and assign the skeleton node to their `Skeleton` property. This will ensure that they can eventually be deformed by it.


Filter nodes
Node2D
polygons
arm_left
leg_left
body
head
jaw
leg_right
arm_right
skeleton
hip
chest
head
jaw
arm_right
hand_right
arm_left
hand_left
arm_left
Filter properties
Polygon2D
Polygon
PoolVector2Arr
Uv
PoolVector2Arr
Splits
PoolIntArray (siz
Color
Vertex Colors
PoolColorArray (
Offset
x 0
y 0
Antialiased
On
Texture
Skeleton
Skeleton
Assign..
Invert
Node2D
Transform
Z Index
CanvasItem

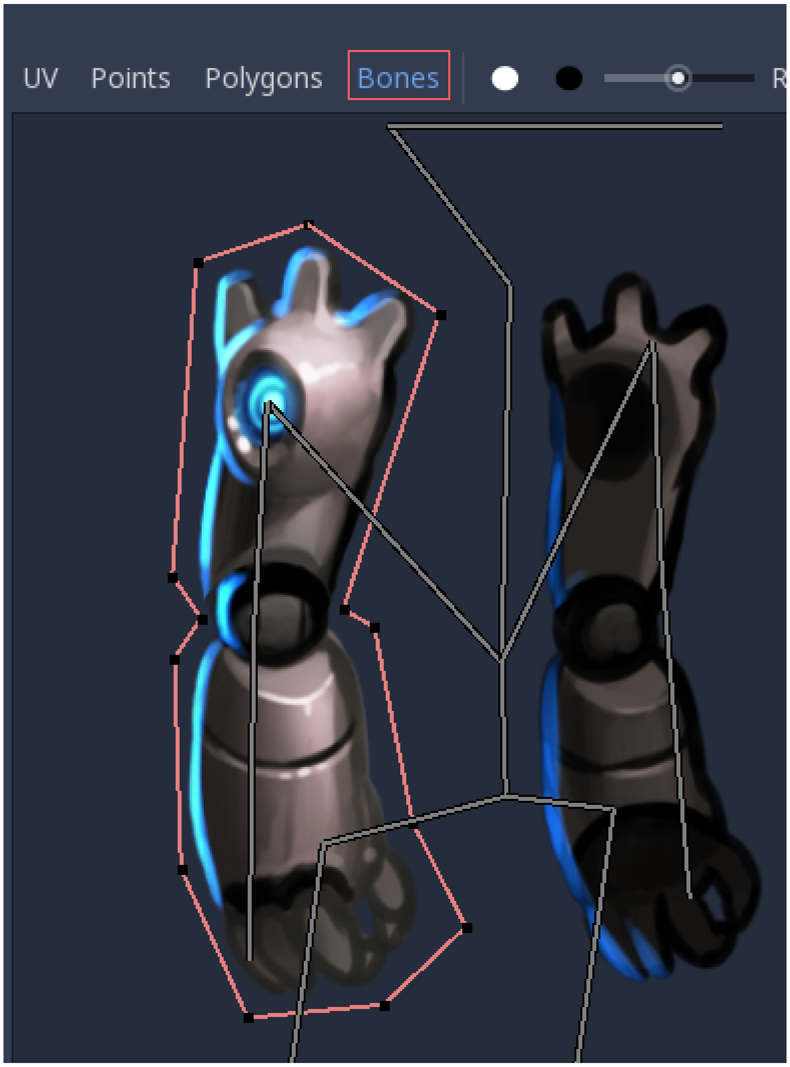
Click the property highlighted above and select the skeleton node:



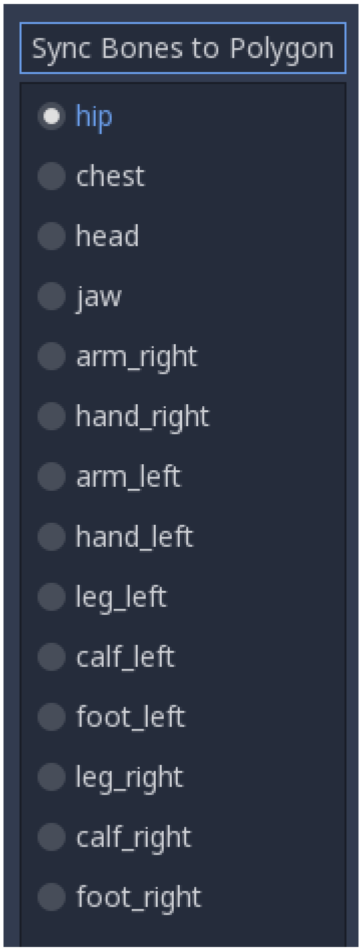
Select a Node

- Node2D
  - polygons
    - arm_left
    - leg_left
    - body
    - head
    - jaw
    - leg_right
    - arm_right
  - skeleton
    - hip
      - chest
        - head
          - jaw
        - arm_right
          - hand_right
        - arm_left
          - hand_left
      - leg_left
        - calf_left
          - foot_left
      - leg_right
        - calf_right
          - foot_right

Again, open the UV editor for the polygon and go to the *Bones* section.

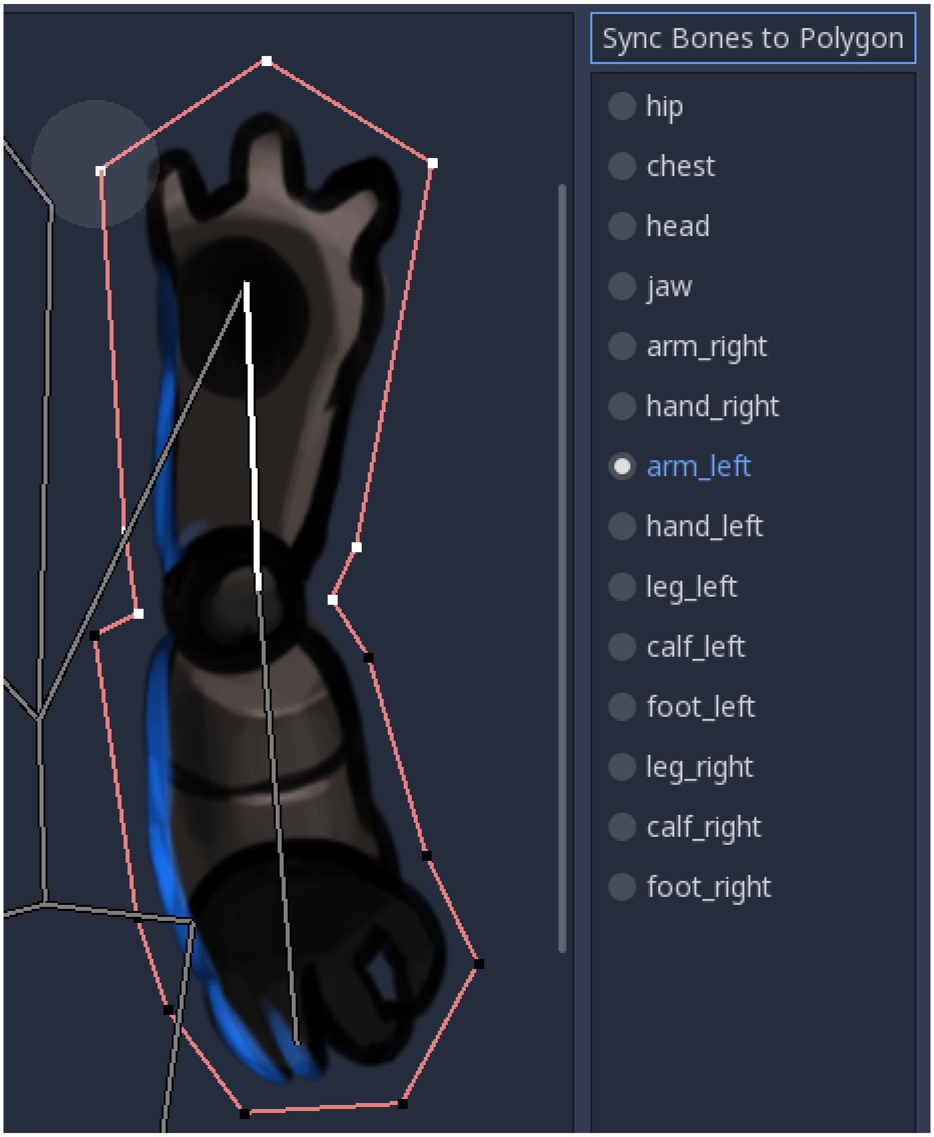


You will not be able to paint weights yet. For this you need to synchronize the list of bones from the skeleton with the polygon. This step is done only once and manually (unless you modify the skeleton by adding/removing/renaming bones). It ensures that your rigging information is kept in the polygon, even if a skeleton node is accidentally lost or the skeleton modified. Push the "Sync Bones to Polygon" button to sync the list.

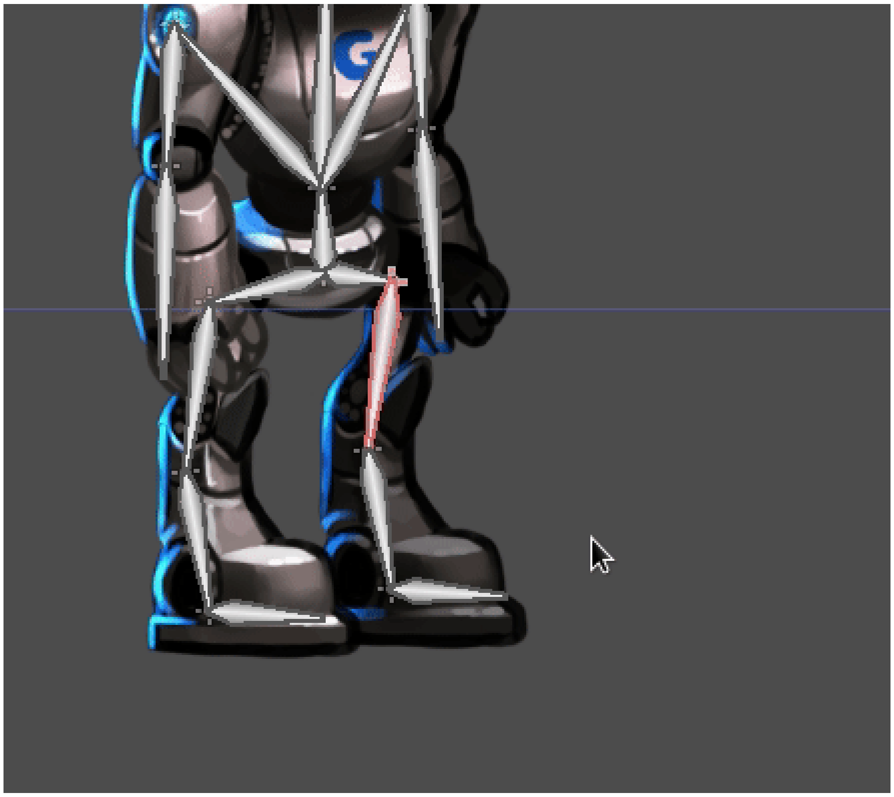


Sync Bones to Polygon
hip
chest
head
jaw
arm_right
hand_right
arm_left
hand_left
leg_left
calf_left
foot_left
leg_right
calf_right
foot_right

The list of bones will automatically appear. By default, your polygon has no weight assigned to any of them. Select the bones you want to assign weight to and paint them:

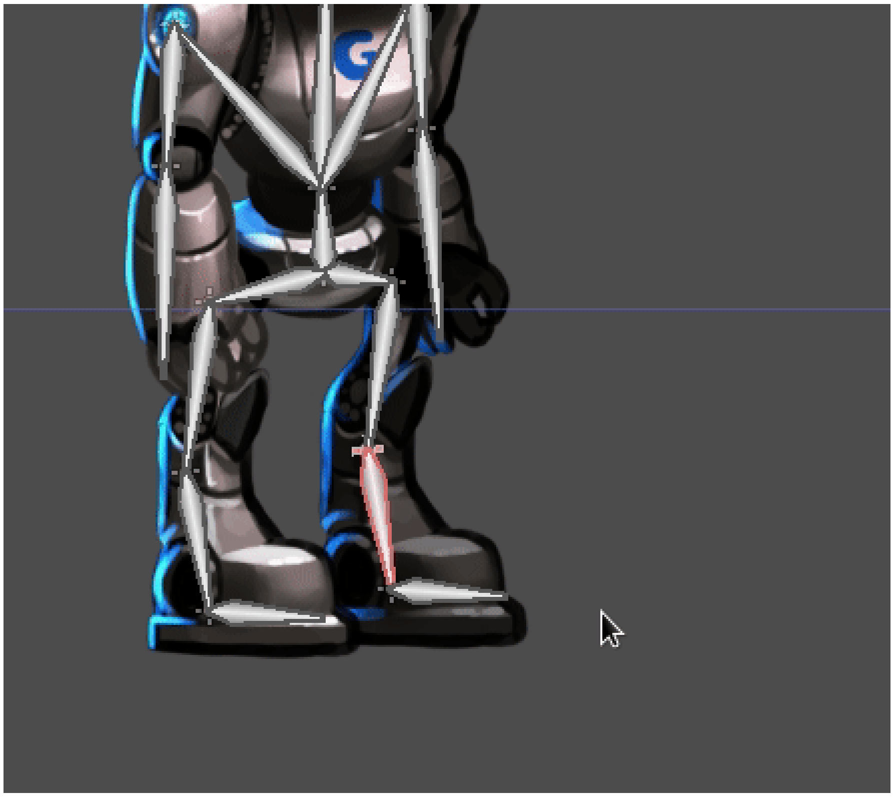

Sync Bones to Polygon
hip
chest
head
jaw
arm_right
hand_right
arm_left
hand_left
leg_left
calf_left
foot_left
leg_right
calf_right
foot_right

Points in white have a full weight assigned, while points in black are not influenced by the bone. If the same point is painted white for multiple bones, the influence will be distributed amongst them (so usually there is not that much need to use shades in-between unless you want to polish the bending effect).



After painting the weights, animating the bones (NOT the polygons!) will have the desired effect of modifying and bending the polygons accordingly. As you only need to animate bones in this approach, work becomes much easier!

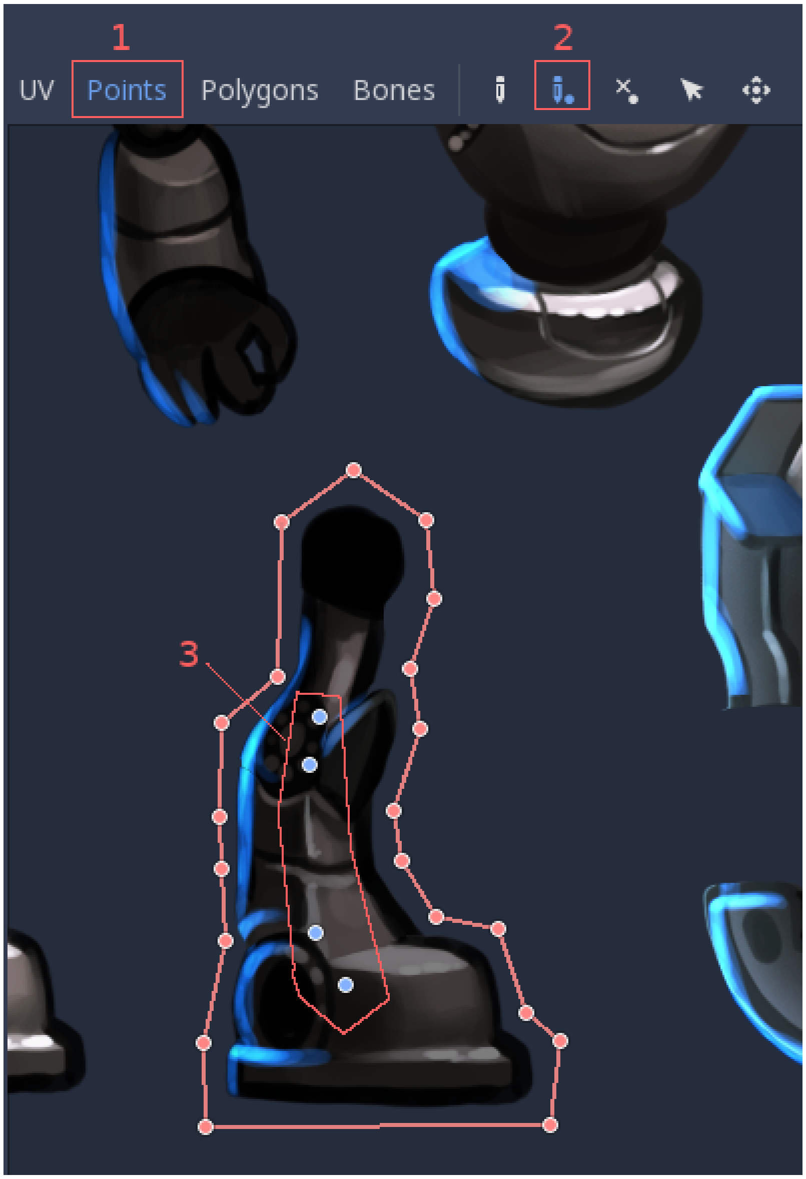
But it's not all roses. Trying to animate bones that bend the polygon will often yield unexpected results:



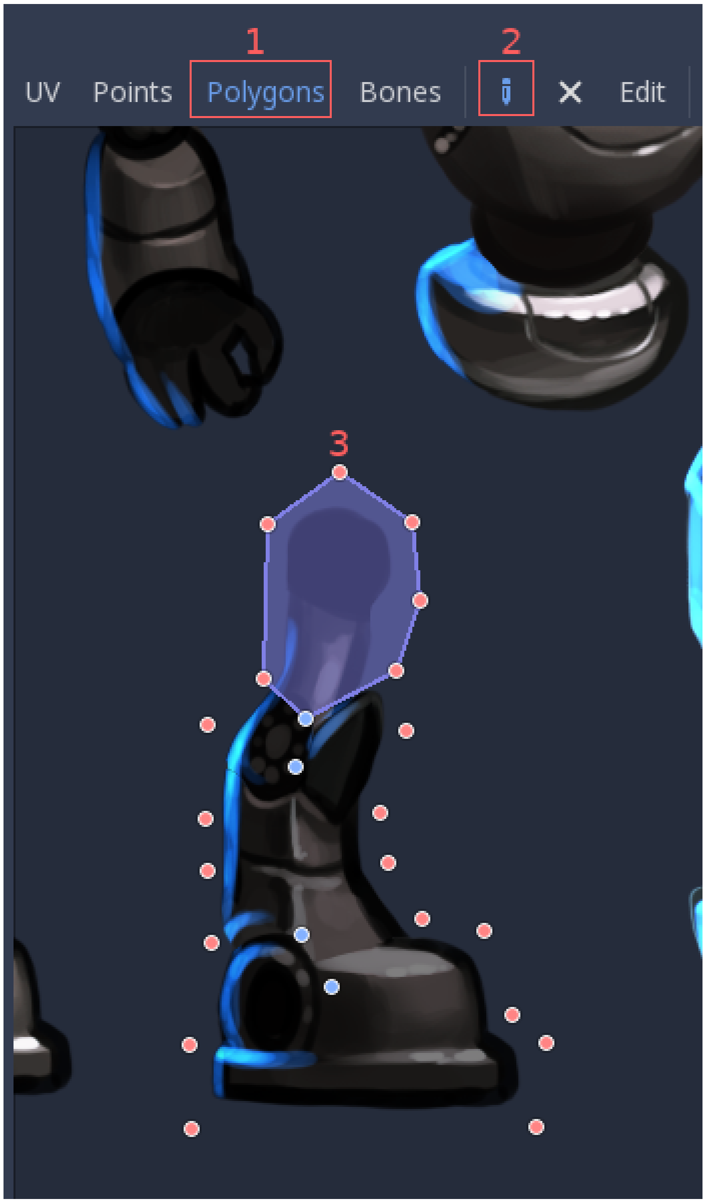
This happens because Godot generates internal triangles that connect the points when drawing the polygon. They don't always bend the way you would expect. To solve this, you need to set hints in the geometry to clarify how you expect it to deform.

Internal vertices

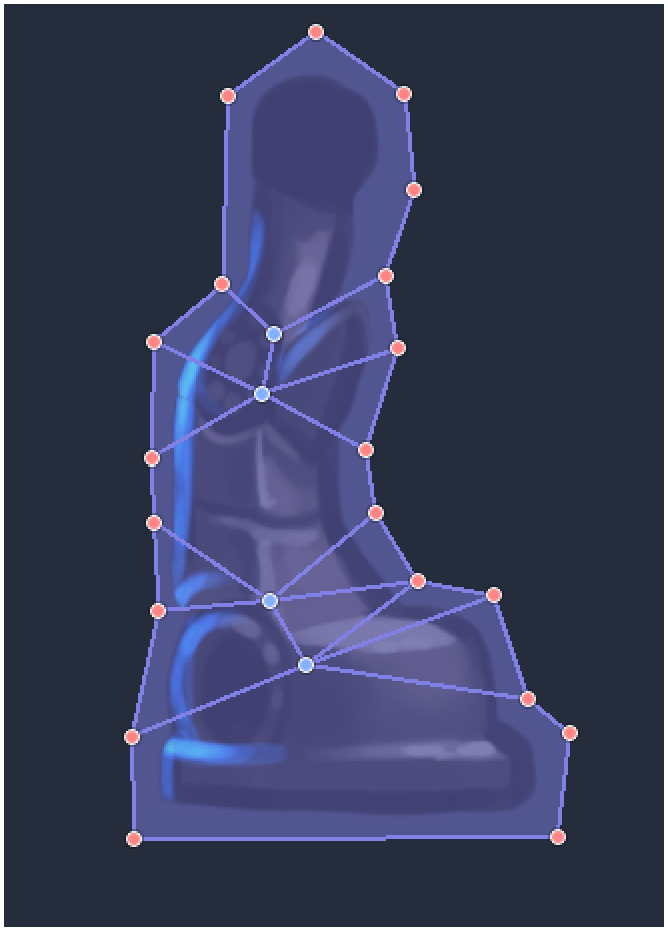
Open the UV menu for each bone again and go to the *Points* section. Add some internal vertices in the regions where you expect the geometry to bend:


1
2
UV
Points
Polygons
Bones
3

Now, go to the *Polygon* section and redraw your own polygons with more detail. Imagine that, as your polygons bend, you need to make sure they deform the least possible, so experiment a bit to find the right setup.


1
2
UV
Points
Polygons
Bones
Edit
3

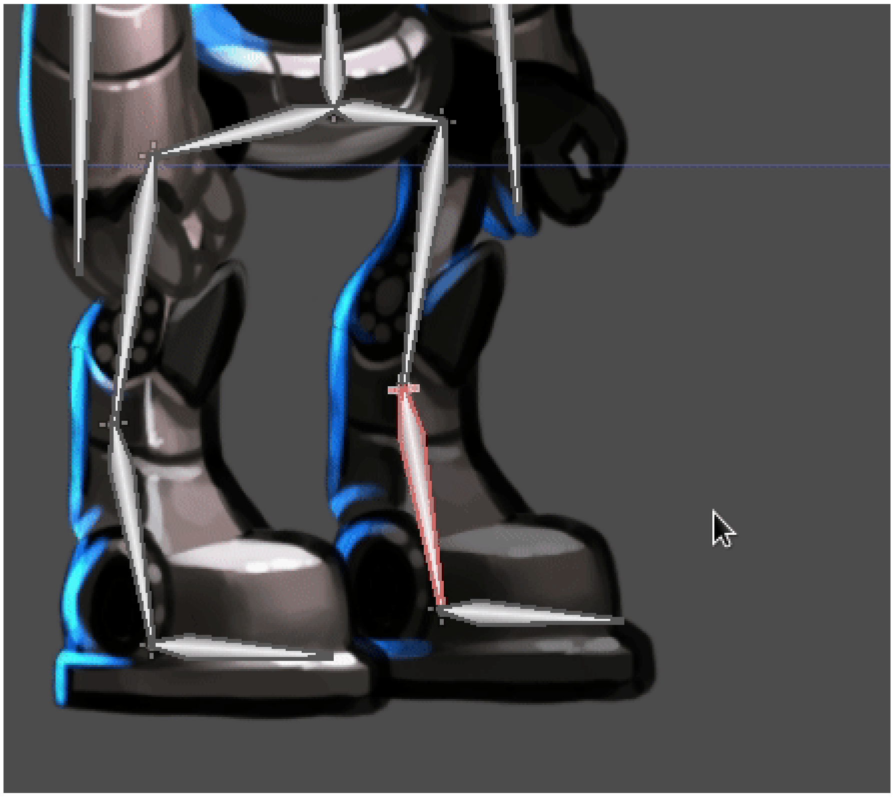
Once you start drawing, the original polygon will disappear and you will be free to create your own:



This amount of detail is usually fine, though you may want to have more fine-grained control over where triangles go. Experiment by yourself until you get the results you like.

Note: Don't forget that your newly added internal vertices also need weight painting! Go to the *Bones* section again to assign them to the right bones.

Once you are all set, you will get much better results:



Using AnimationTree

Introduction

With AnimationPlayer, Godot has one of the most flexible animation systems that you can find in any game engine. The ability to animate almost any property in any node or resource, as well as having dedicated transform, bezier, function calling, audio and sub-animation tracks, is pretty much unique.

However, the support for blending those animations via `AnimationPlayer` is relatively limited, as only a fixed cross-fade transition time can be set.

AnimationTree is a new node introduced in Godot 3.1 to deal with advanced transitions. It supersedes the ancient `AnimationTreePlayer`, while adding a huge amount of features and flexibility.

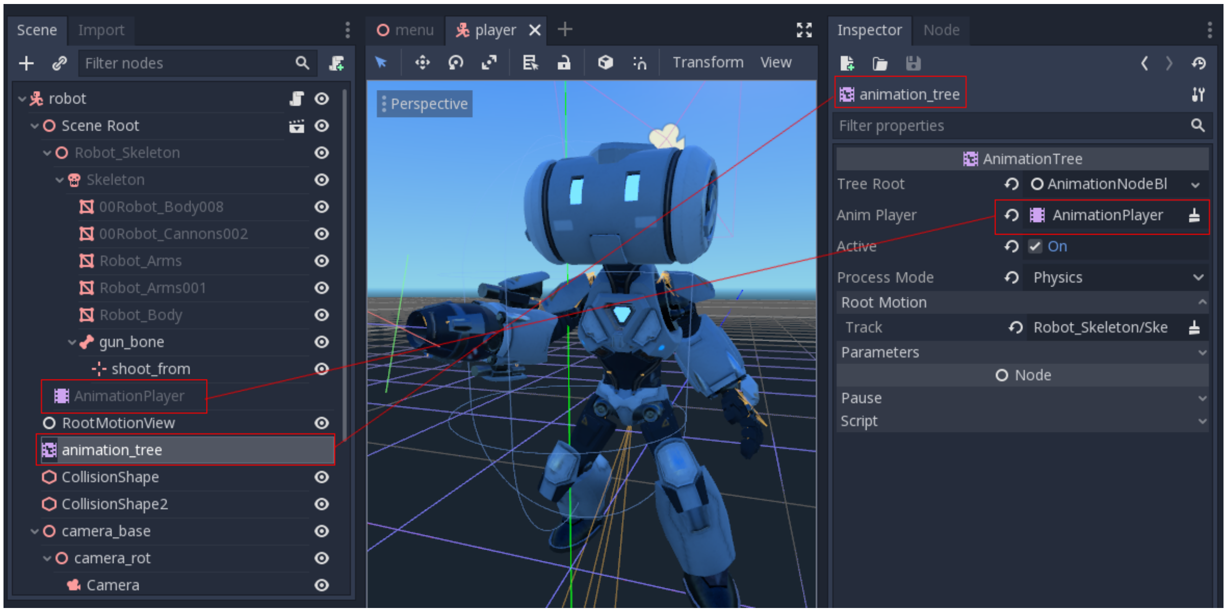
Creating an AnimationTree

Before starting, it must be made clear that an `AnimationTree` node does not contain its own animations. Instead, it uses animations contained in an `AnimationPlayer` node. This way, you can edit your animations (or import them from a 3D scene) as usual and then use this extra node to control the playback.

The most common way to use `AnimationTree` is in a 3D scene. When importing your scenes from a 3D exchange format, they will usually come with animations built-in (either multiple ones or split from a large one on import). At the end, the imported Godot scene will contain the animations in a `AnimationPlayer` node.

As you rarely use imported scenes directly in Godot (they are either instantiated or inherited from), you can place the `AnimationTree` node in your new scene which contains the imported one. Afterwards, point the `AnimationTree` node to the `AnimationPlayer` that was created in the imported scene.

This is how it's done in the [Third Person Shooter demo](https://github.com/godotengine/tps-demo) [https://github.com/godotengine/tps-demo], for reference:



Scene
Import
Filter nodes
robot
Scene Root
Robot_Skeleton
Skeleton
00Robot_Body008
00Robot_Cannons002
Robot_Arms
Robot_Arms001
Robot_Body
gun_bone
shoot_from
AnimationPlayer
RootMotionView
animation_tree
CollisionShape
CollisionShape2
camera_base
camera_rot
Camera
menu
player
Transform
View
Perspective
Inspector
Node
animation_tree
Filter properties
AnimationTree
Root Motion
Parameters
Node
Pause
Script

Tree Root: AnimationNodeBl
Anim Player: AnimationPlayer
Active: ☑ On
Process Mode: Physics
Track: Robot_Skeleton/Ske

A new scene was created for the player with a `CharacterBody3D` as root. Inside this scene, the original `.dae` (Collada) file was instantiated and an `AnimationTree` node was created.

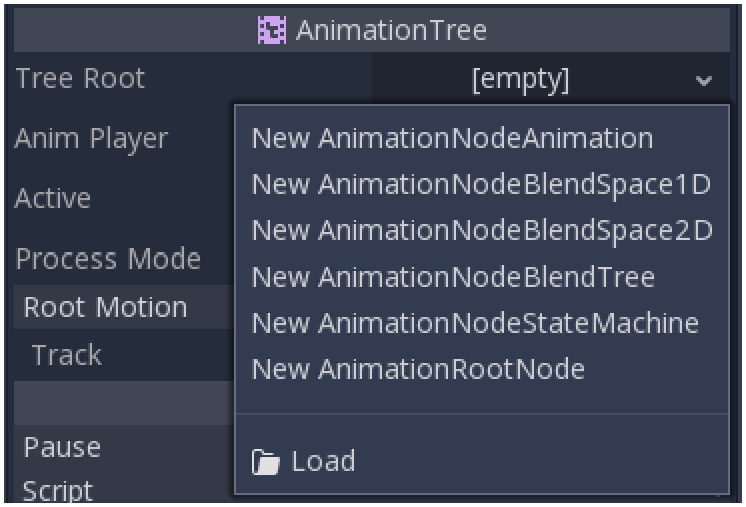
Creating a tree

There are three main types of nodes that can be used in `AnimationTree`:

1. Animation nodes, which reference an animation from the linked `AnimationPlayer`.
2. Animation Root nodes, which are used to blend sub-nodes.

3. Animation Blend nodes, which are used within `AnimationNodeBlendTree` as single-graph blending via multiple input ports.

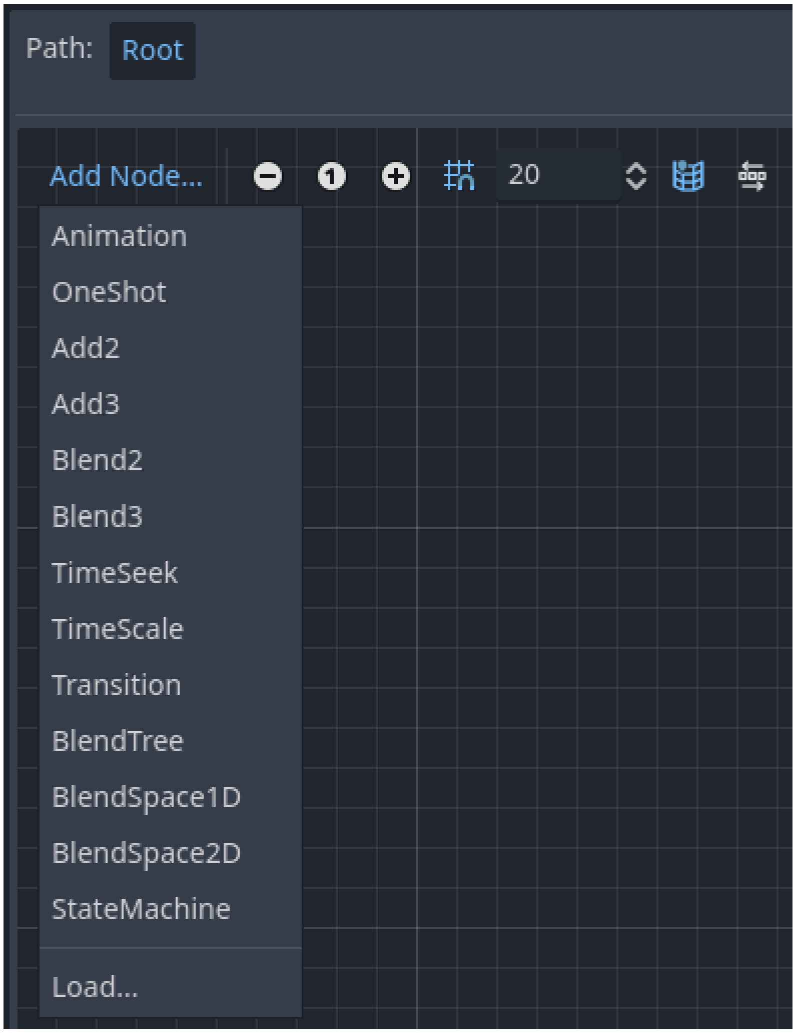
To set a root node in `AnimationTree`, a few types are available:


AnimationTree
Tree Root
[empty]
Anim Player
Active
Process Mode
Root Motion
Track
Pause
Script
New AnimationNodeAnimation
New AnimationNodeBlendSpace1D
New AnimationNodeBlendSpace2D
New AnimationNodeBlendTree
New AnimationNodeStateMachine
New AnimationRootNode
Load

- `AnimationNodeAnimation`: Selects an animation from the list and plays it. This is the simplest root node, and generally not used directly as root.
- `AnimationNodeBlendTree`: Contains many *blend* type nodes, such as mix, blend2, blend3, one shot, etc. This is one of the most commonly used roots.
- `AnimationNodeStateMachine`: Contains multiple root nodes as children in a graph. Each node is used as a *state*, and provides multiple functions to alternate between states.
- `AnimationNodeBlendSpace2D`: Allows placing root nodes in a 2D blend space. Control the blend position in 2D to mix between multiple animations.
- `AnimationNodeBlendSpace1D`: Simplified version of the above (1D).

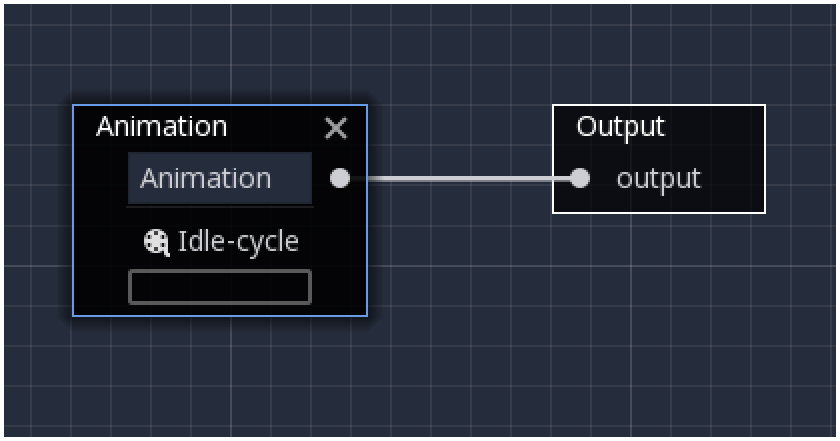
Blend tree

An `AnimationNodeBlendTree` can contain both root and regular nodes used for blending. Nodes are added to the graph from a menu:


Path:
Root
Add Node...
20
Animation
OneShot
Add2
Add3
Blend2
Blend3
TimeSeek
TimeScale
Transition
BlendTree
BlendSpace1D
BlendSpace2D
StateMachine
Load...

All blend trees contain an `Output` node by default, and something has to be connected to it in order for animations to play.

The easiest way to test this functionality is to connect an `Animation` node to it directly:

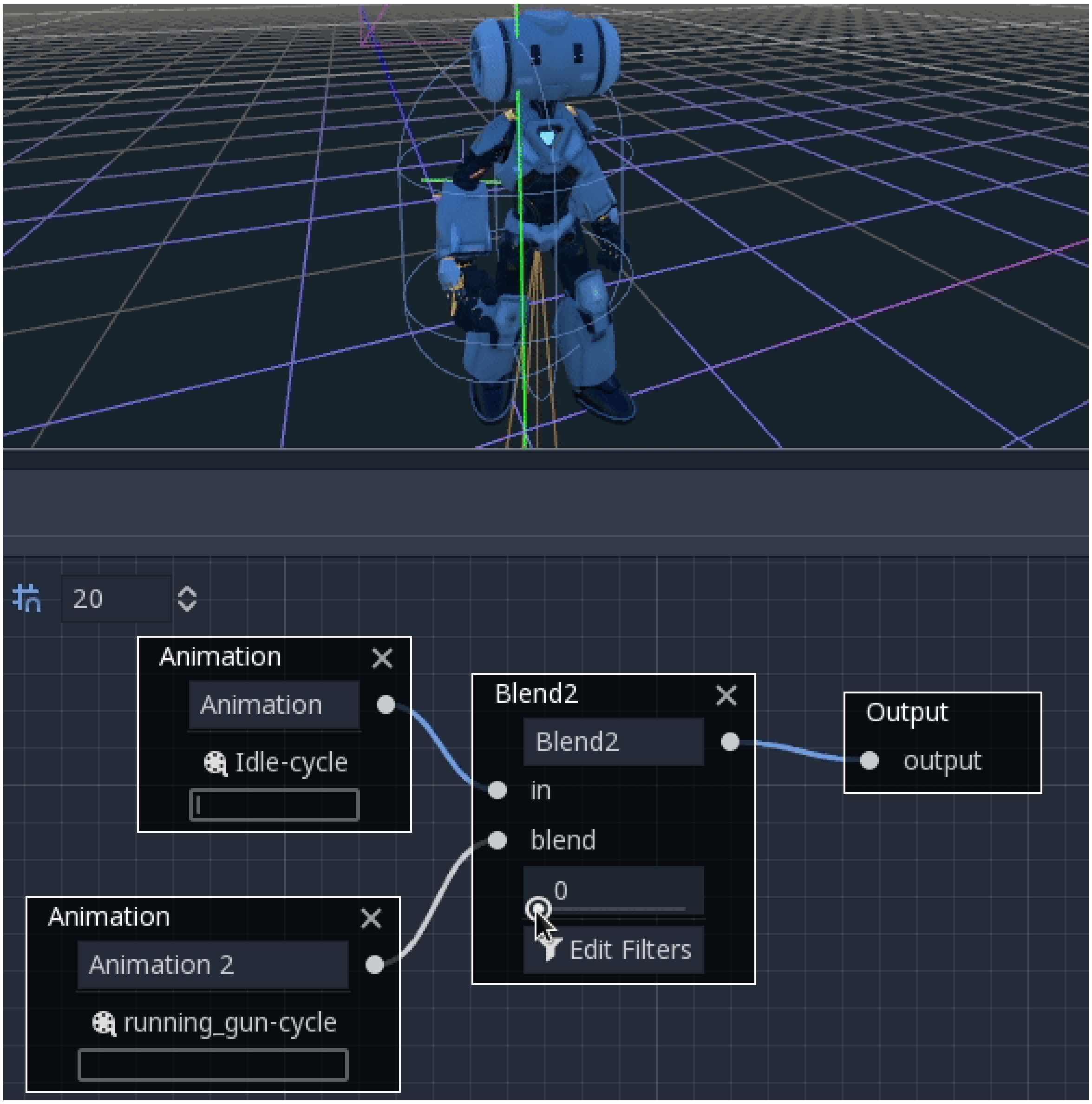

Animation
Animation
Idle-cycle
Output
output

This will simply play back the animation. Make sure that the `AnimationTree` is active for something to actually happen.

Following is a short description of available nodes:

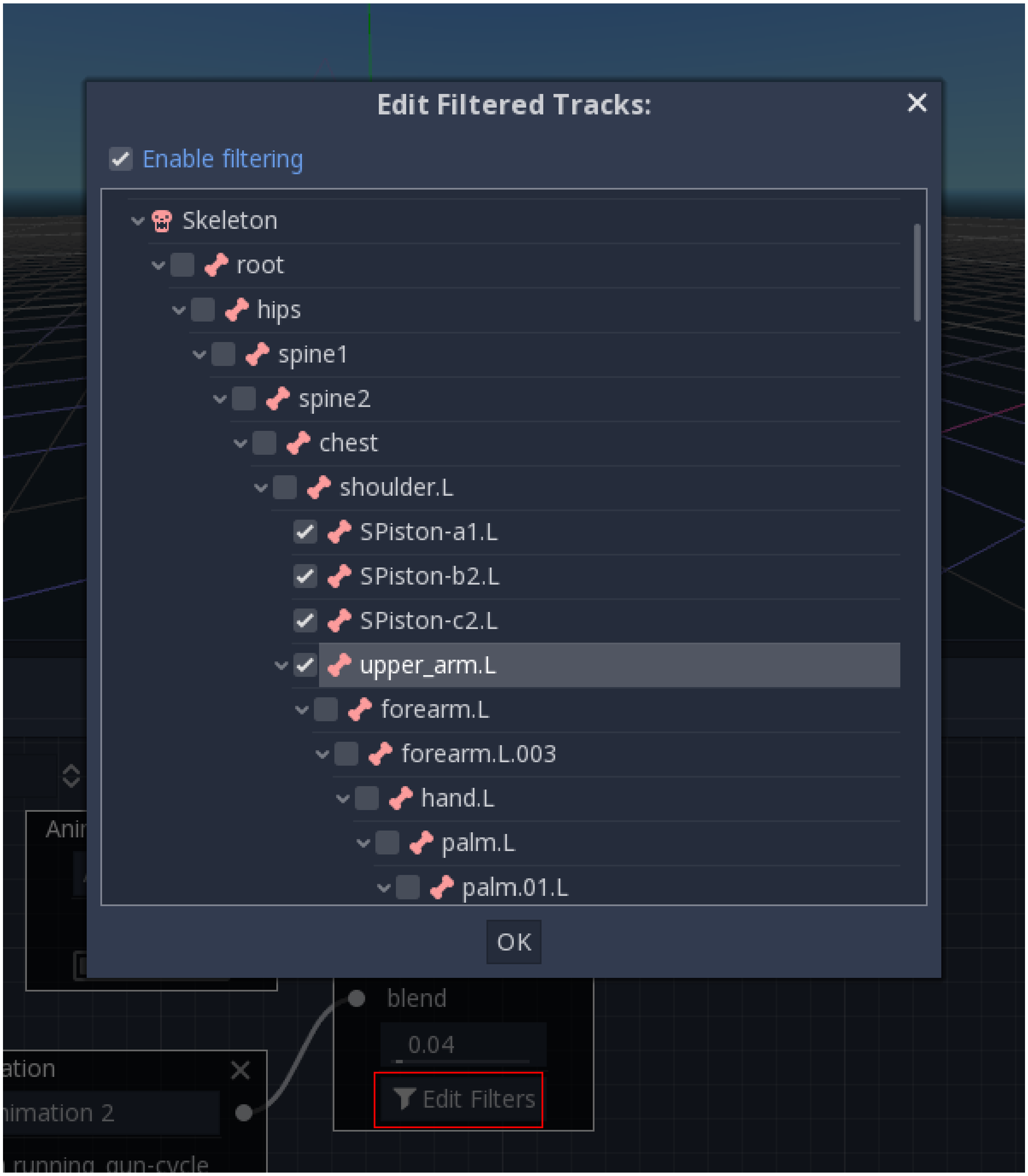
Blend2 / Blend3

These nodes will blend between two or three inputs by a user-specified blend value:


20
Animation
Animation
Idle-cycle
Blend2
Blend2
in
blend
0
Edit Filters
Output
output
Animation
Animation 2
running_gun-cycle

For more complex blending, it is advised to use blend spaces instead.

Blending can also use filters, i.e. you can control individually which tracks go through the blend function. This is very useful for layering animations on top of each other.



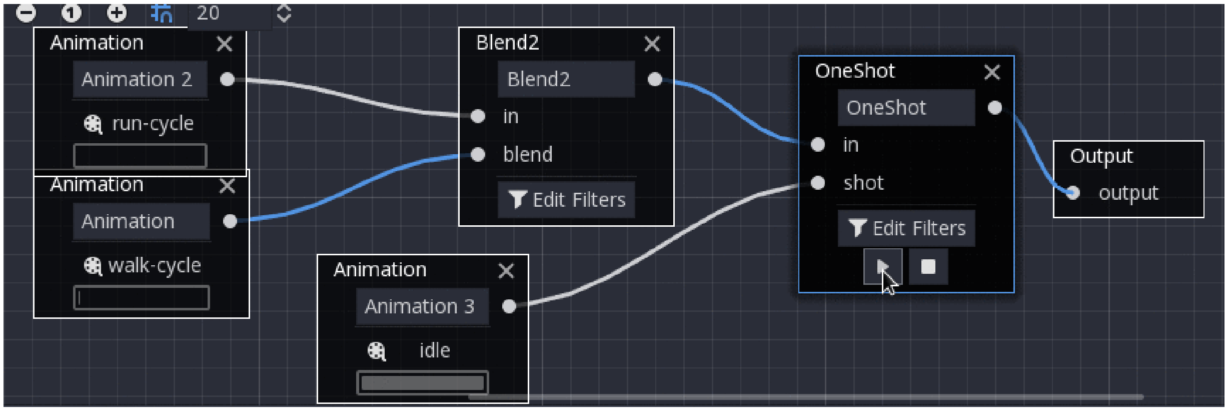
Edit Filtered Tracks:
OK
blend
0.04
Edit Filters
ation
imation 2

☑ Enable filtering

- Skeleton
  - ☐ root
    - ☐ hips
      - ☐ spine1
        - ☐ spine2
          - ☐ chest
            - ☐ shoulder.L
              - ☑ SPiston-a1.L
              - ☑ SPiston-b2.L
              - ☑ SPiston-c2.L
              - ☑ upper_arm.L
                - ☐ forearm.L
                  - ☐ forearm.L.003
                    - ☐ hand.L
                      - ☐ palm.L
                        - ☐ palm.01.L

OneShot

This node will execute a sub-animation and return once it finishes. Blend times for fading in and out can be customized, as well as filters.


20
Animation
Animation 2
run-cycle
Animation
Animation
walk-cycle
Blend2
Blend2
in
blend
Edit Filters
Animation
Animation 3
idle
OneShot
OneShot
in
shot
Edit Filters
Output
output

After setting the request and changing the animation playback, the one-shot node automatically clears the request on the next process frame by setting its `request` value to `AnimationNodeOneShot.ONE_SHOT_REQUEST_NONE`.

GDScript

```
# Play child animation connected to "shot" port.
animation_tree.set("parameters/OneShot/request",
AnimationNodeOneShot.ONE_SHOT_REQUEST_FIRE)
# Alternative syntax (same result as above).
animation_tree["parameters/OneShot/request"] =
AnimationNodeOneShot.ONE_SHOT_REQUEST_FIRE

# Abort child animation connected to "shot" port.
animation_tree.set("parameters/OneShot/request",
AnimationNodeOneShot.ONE_SHOT_REQUEST_ABORT)
# Alternative syntax (same result as above).
animation_tree["parameters/OneShot/request"] =
AnimationNodeOneShot.ONE_SHOT_REQUEST_ABORT

# Get current state (read-only).
animation_tree.get("parameters/OneShot/active"))
# Alternative syntax (same result as above).
animation_tree["parameters/OneShot/active"]
```

C#

```
// Play child animation connected to "shot" port.
animationTree.Set("parameters/OneShot/request",
(int)AnimationNodeOneShot.OneShotRequest.Fire);

// Abort child animation connected to "shot" port.
animationTree.Set("parameters/OneShot/request",
(int)AnimationNodeOneShot.OneShotRequest.Abort);

// Get current state (read-only).
animationTree.Get("parameters/OneShot/active");
```

TimeSeek

This node can be used to cause a seek command to happen to any sub-children of the animation graph. Use this node type to play an `Animation` from the start or a certain playback position inside the `AnimationNodeBlendTree`.

After setting the time and changing the animation playback, the seek node automatically goes into sleep mode on the next process frame by setting its `seek_request` value to `-1.0`.

GDScript

```
# Play child animation from the start.
animation_tree.set("parameters/TimeSeek/seek_request", 0.0)
# Alternative syntax (same result as above).
animation_tree["parameters/TimeSeek/seek_request"] = 0.0

# Play child animation from 12 second timestamp.
animation_tree.set("parameters/TimeSeek/seek_request", 12.0)
# Alternative syntax (same result as above).
animation_tree["parameters/TimeSeek/seek_request"] = 12.0
```

C#

```
// Play child animation from the start.
animationTree.Set("parameters/TimeSeek/seek_request", 0.0);
```

```
// Play child animation from 12 second timestamp.
animationTree.Set("parameters/TimeSeek/seek_request", 12.0);
```

TimeScale

Allows scaling the speed of the animation (or reverse it) connected to the *in* input via the *scale* parameter. Setting the *scale* to 0 will pause the animation.

Transition

Very simple state machine (when you don't want to cope with a `StateMachine` node). Animations can be connected to the outputs and transition times can be specified. After setting the request and changing the animation playback, the transition node automatically clears the request on the next process frame by setting its `transition_request` value to an empty string (`""`).

GDScript

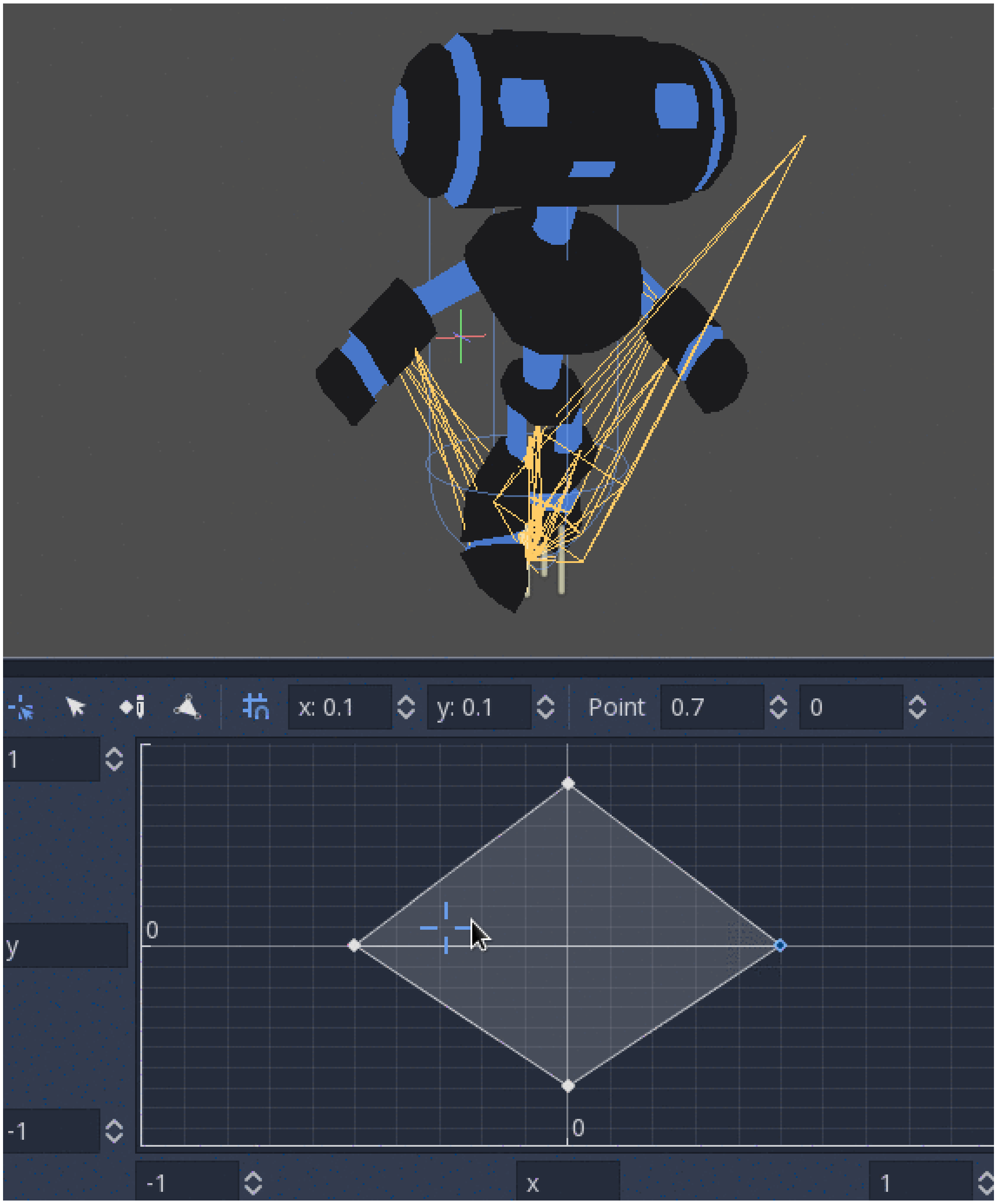
```
# Play child animation connected to "state_2" port.
animation_tree.set("parameters/Transition/transition_request",
"state_2")
# Alternative syntax (same result as above).
animation_tree["parameters/Transition/transition_request"] =
"state_2"

# Get current state name (read-only).
animation_tree.get("parameters/Transition/current_state")
# Alternative syntax (same result as above).
animation_tree["parameters/Transition/current_state"]

# Get current state index (read-only).
animation_tree.get("parameters/Transition/current_index"))
# Alternative syntax (same result as above).
animation_tree["parameters/Transition/current_index"]
```

C#

```
// Play child animation connected to "state_2" port.
animationTree.Set("parameters/Transition/transition_request",
"state_2");

// Get current state name (read-only).
animationTree.Get("parameters/Transition/current_state");

// Get current state index (read-only).
animationTree.Get("parameters/Transition/current_index");
```

BlendSpace2D

`BlendSpace2D` is a node to do advanced blending in two dimensions. Points are added to a two-dimensional space and then a position can be controlled to determine blending:

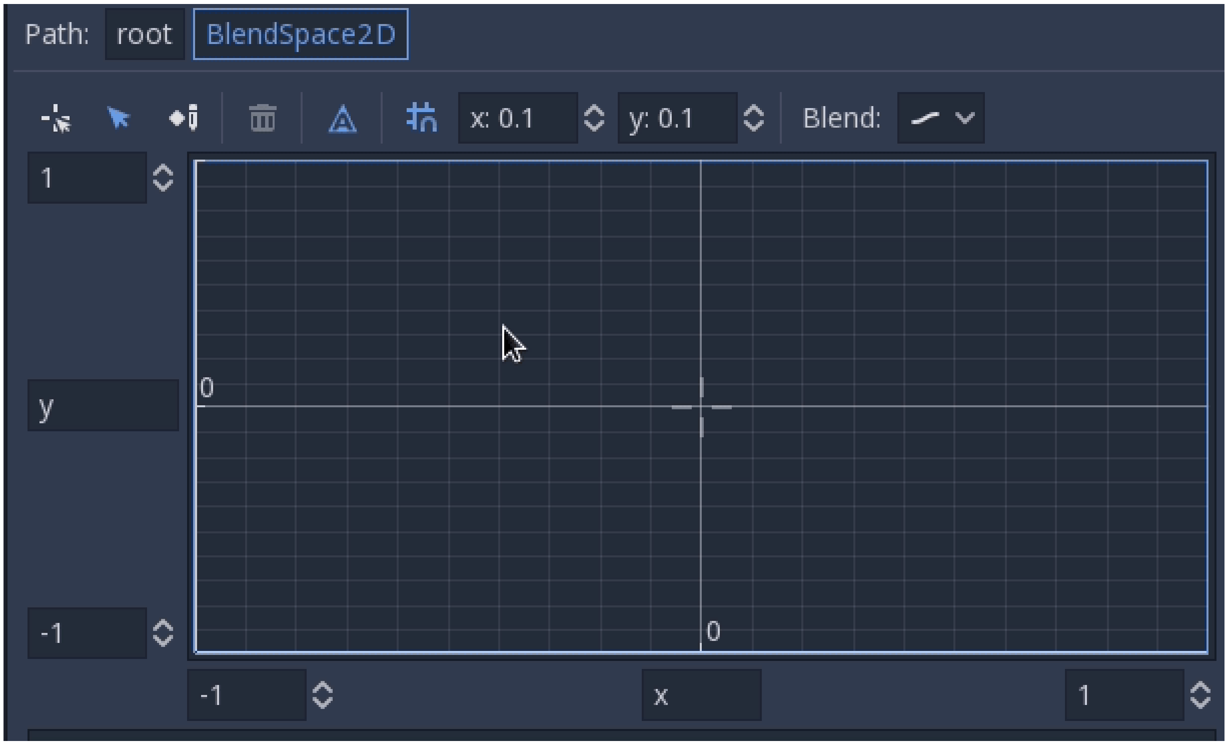


x: 0.1

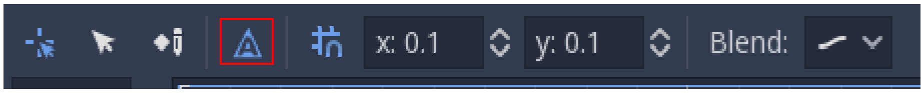
y: 0.1

Point 0.7 0

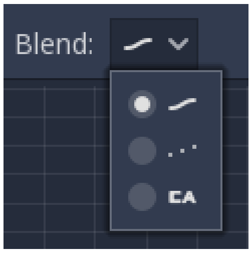
The ranges in X and Y can be controlled (and labeled for convenience). By default, points can be placed anywhere (right-click on the coordinate system or use the *add point* button) and triangles will be generated automatically using Delaunay.


Path: root BlendSpace2D
x: 0.1
y: 0.1
Blend:
1
y
-1
0
0
-1
x
1

It is also possible to draw the triangles manually by disabling the *auto triangle* option, though this is rarely necessary:


x: 0.1
y: 0.1
Blend:

Finally, it is possible to change the blend mode. By default, blending happens by interpolating points inside the closest triangle. When dealing with 2D animations (frame by frame), you may want to switch to *Discrete* mode. Alternatively, if you want to keep the current play position when switching between discrete animations, there is a *Carry* mode. This mode can be changed in the *Blend* menu:

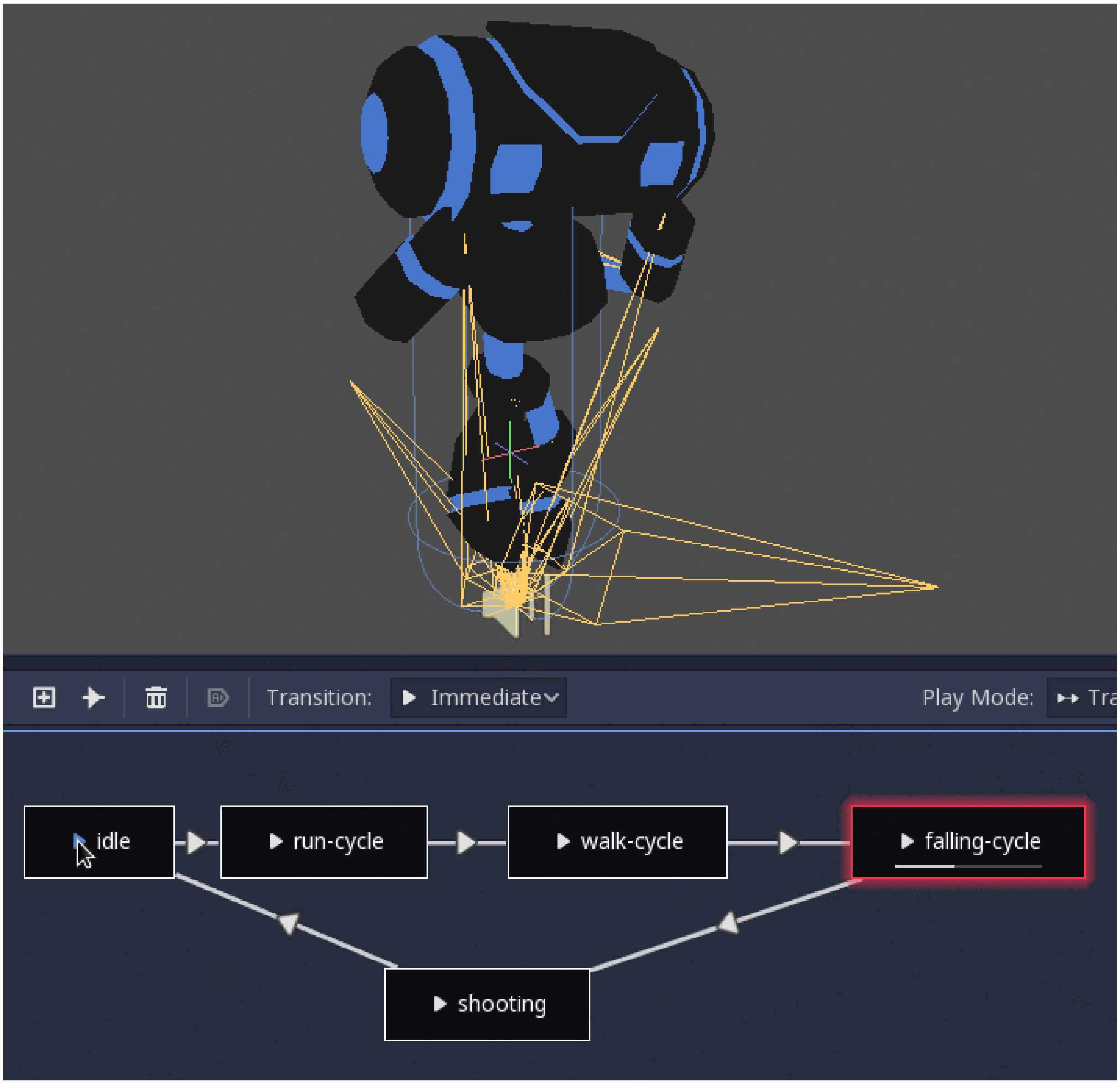


BlendSpace1D

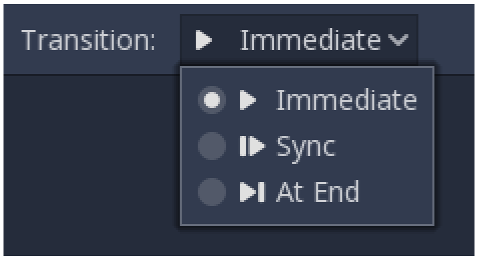
This is similar to 2D blend spaces, but in one dimension (so triangles are not needed).

StateMachine

This node acts as a state machine with root nodes as states. Root nodes can be created and connected via lines. States are connected via *Transitions*, which are connections with special properties. Transitions are uni-directional, but two can be used to connect in both directions.


Transition:
Immediate
Play Mode:
Tra
idle
run-cycle
walk-cycle
falling-cycle
shooting

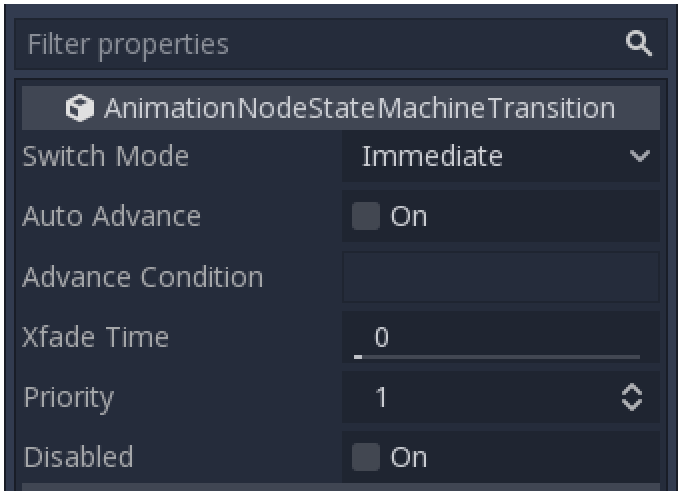
There are many types of transition:


Transition:
Immediate
Immediate
Sync
At End

- *Immediate*: Will switch to the next state immediately. The current state will end and blend into the beginning of the new

one.
- *Sync*: Will switch to the next state immediately, but will seek the new state to the playback position of the old state.
- *At End*: Will wait for the current state playback to end, then switch to the beginning of the next state animation.

Transitions also have a few properties. Click any transition and it will be displayed in the inspector dock:



Switch Mode: Immediate
Auto Advance: ☐ On
Advance Condition:
Xfade Time: 0
Priority: 1
Disabled: ☐ On

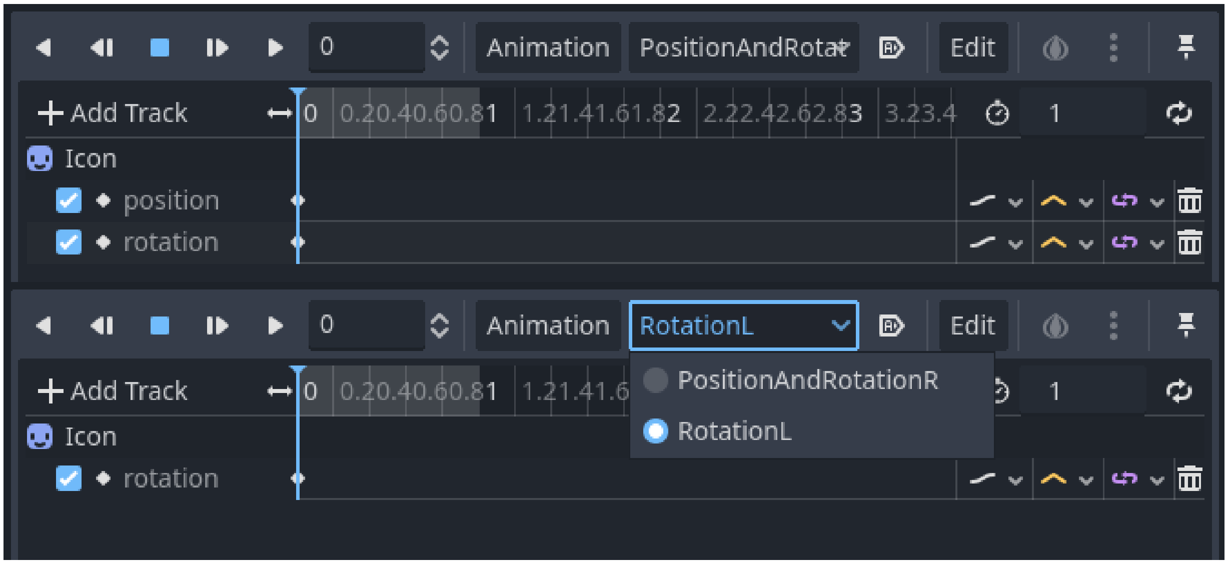
- *Switch Mode* is the transition type (see above), it can be modified after creation here.
- *Auto Advance* will turn on the transition automatically when this state is reached. This works best with the *At End* switch mode.
- *Advance Condition* will turn on auto advance when this condition is set. This is a custom text field that can be filled with a variable name. The variable can be modified from code (more on this later).
- *Xfade Time* is the time to cross-fade between this state and the next.
- *Priority* is used together with the `travel()` function from code (more on this later). Lower priority transitions are preferred when travelling through the tree.
- *Disabled* toggles disabling this transition (when disabled, it will not be used during travel or auto advance).

For better blending

In Godot 4.0+, in order for the blending results to be deterministic (reproducible and always consistent), the blended property values must have a specific initial value. For example, in the case of two animations to be blended, if one animation has a property track and the other does not, the blended animation is calculated as if the latter animation had a property track with the initial value.

When using Position/Rotation/Scale 3D tracks for Skeleton3D bones, the initial value is Bone Rest. For other properties, the initial value is `0` and if the track is present in the `RESET` animation, the value of its first keyframe is used instead.

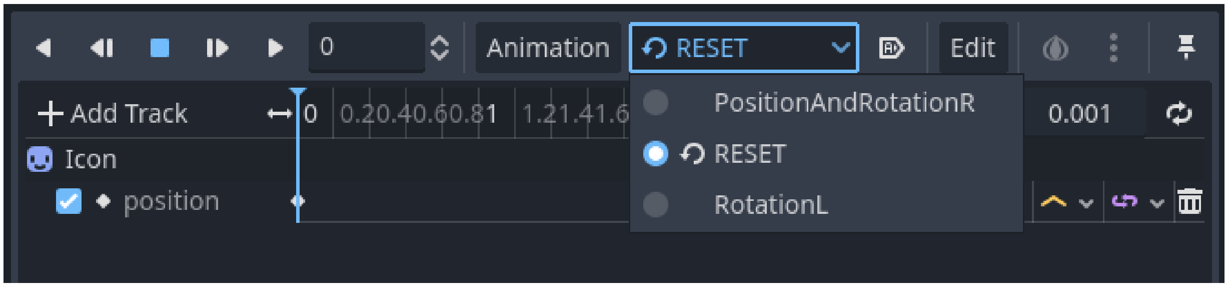
For example, the following AnimationPlayer has two animations, but one of them lacks a Property track for Position.


0
Animation
PositionAndRotat
Edit
Add Track
0 0.20.40.60.81 1.21.41.61.82 2.22.42.62.83 3.23.4
1
Icon
position
rotation
0
Animation
RotationL
Edit
Add Track
0 0.20.40.60.81 1.21.41.6
1
PositionAndRotationR
RotationL
Icon
rotation

This means that the animation lacking that will treat those Positions as `Vector2(0, 0)`.

tutorials/animation/img/blending2.png

This problem can be solved by adding a Property track for Position as an initial value to the `RESET` animation.



tutorials/animation/img/blending4.png

Note

Be aware that the `RESET` animation exists to define the default pose when loading an object originally. It is assumed to have only one frame and is not expected to be played back using the timeline.

Also keep in mind that the Rotation 3D tracks and the Property tracks for 2D rotation with Interpolation Type set to Linear Angle or Cubic Angle will prevent rotation of more than 180 degrees from the initial value as blended animation.

This can be useful for Skeleton3Ds to prevent the bones penetrating the body when blending animations. Therefore, Skeleton3D's Bone Rest values should be as close to the midpoint of the movable range as possible. **This means that for humanoid models, it is preferable to import them in a T-pose**.

tutorials/animation/img/blending5.png

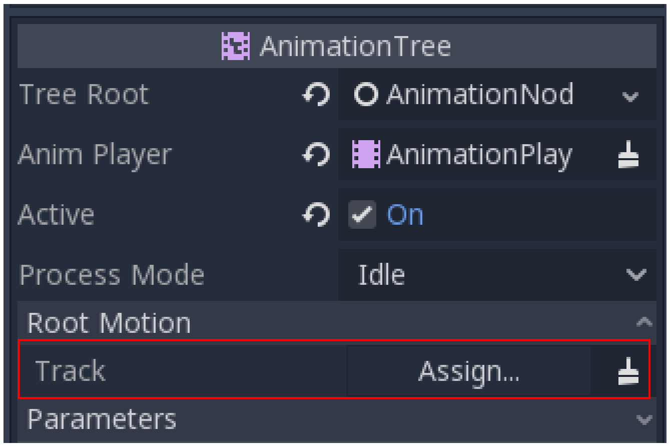
You can see that the shortest rotation path from Bone Rests is prioritized rather than the shortest rotation path between animations.

If you need to rotate Skeleton3D itself more than 180 degrees by blend animations for movement, you can use Root Motion.

Root motion

When working with 3D animations, a popular technique is for animators to use the root skeleton bone to give motion to the rest of the skeleton. This allows animating characters in a way where steps actually match the floor below. It also allows precise interaction with objects during cinematics.

When playing back the animation in Godot, it is possible to select this bone as the *root motion track*. Doing so will cancel the bone transformation visually (the animation will stay in place).



Tree Root: AnimationNod
Anim Player: AnimationPlay
Active: ☑ On
Process Mode: Idle
Track: Assign...

Afterwards, the actual motion can be retrieved via the AnimationTree API as a transform:

GDScript

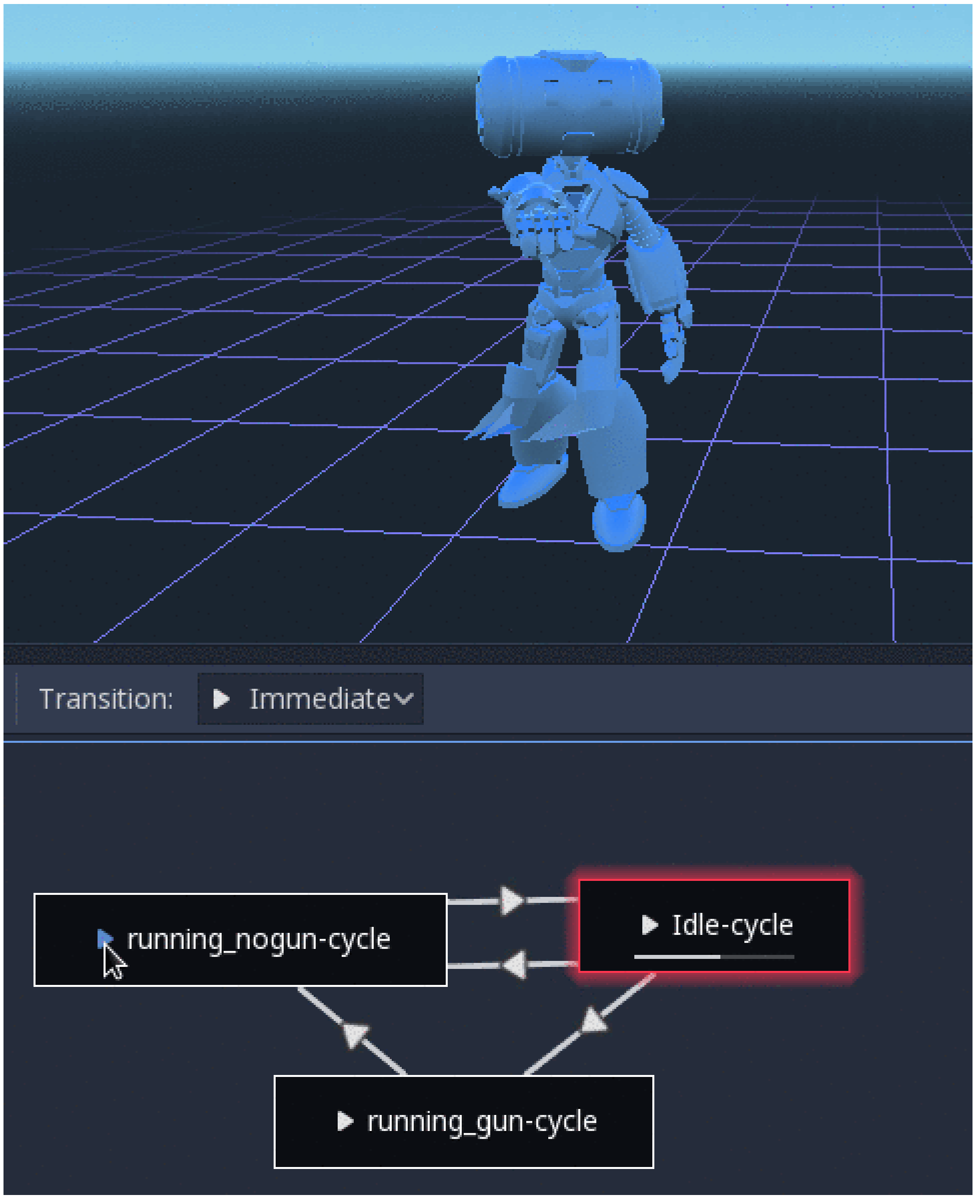
```
# Get the motion delta.
animation_tree.get_root_motion_position()
animation_tree.get_root_motion_rotation()
animation_tree.get_root_motion_scale()

# Get the actual blended value of the animation.
animation_tree.get_root_motion_position_accumulator()
animation_tree.get_root_motion_rotation_accumulator()
animation_tree.get_root_motion_scale_accumulator()
```

C#

```
// Get the motion delta.
animationTree.GetRootMotionPosition();
animationTree.GetRootMotionRotation();
animationTree.GetRootMotionScale();

// Get the actual blended value of the animation.
animationTree.GetRootMotionPositionAccumulator();
animationTree.GetRootMotionRotationAccumulator();
animationTree.GetRootMotionScaleAccumulator();
```

This can be fed to functions such as CharacterBody3D.move_and_slide to control the character movement.

There is also a tool node, `RootMotionView`, that can be placed in a scene and will act as a custom floor for your character and animations (this node is disabled by default during the game).

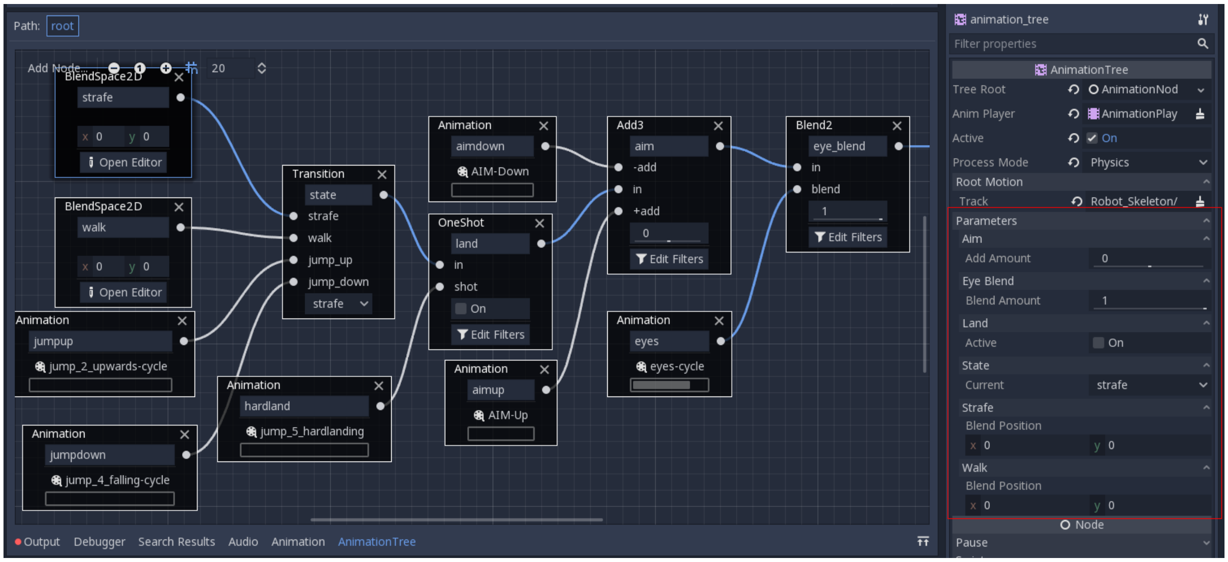

Transition:
Immediate
running_nogun-cycle
Idle-cycle
running_gun-cycle

Controlling from code

After building the tree and previewing it, the only question remaining is "How is all this controlled from code?".

Keep in mind that the animation nodes are just resources and, as such, they are shared between all instances using them. Setting values in the nodes directly will affect all instances of the scene that uses this `AnimationTree`. This is generally undesirable, but does have some cool use cases, e.g. you can copy and paste parts of your animation tree, or reuse nodes with a complex layout (such as a state machine or blend space) in different animation trees.

The actual animation data is contained in the `AnimationTree` node and is accessed via properties. Check the "Parameters" section of the `AnimationTree` node to see all the parameters that can be modified in real-time:


Path: root
Add Node
20
BlendSpace2D
strafe
x 0 y 0
Open Editor
BlendSpace2D
walk
x 0 y 0
Open Editor
Animation
jumpup
jump_2_upwards-cycle
Animation
jumpdown
jump_4_falling-cycle
Transition
state
strafe
walk
jump_up
jump_down
strafe
Animation
hardland
jump_5_hardlanding
Animation
aimdown
AIM-Down
OneShot
land
in
shot
On
Edit Filters
Animation
aimup
AIM-Up
Add3
aim
-add
in
+add
0
Edit Filters
Animation
eyes
eyes-cycle
Blend2
eye_blend
in
blend
1
Edit Filters
Output
Debugger
Search Results
Audio
Animation
AnimationTree

animation_tree

Filter properties

AnimationTree

Tree Root: AnimationNod

Anim Player: AnimationPlay

Active: ☑ On

Process Mode: Physics

Root Motion

Track: Robot_Skeleton/

Parameters

Aim

Add Amount: 0

Eye Blend

Blend Amount: 1

Land

Active: ☐ On

State

Current: strafe

Strafe

Blend Position: x 0 y 0

Walk

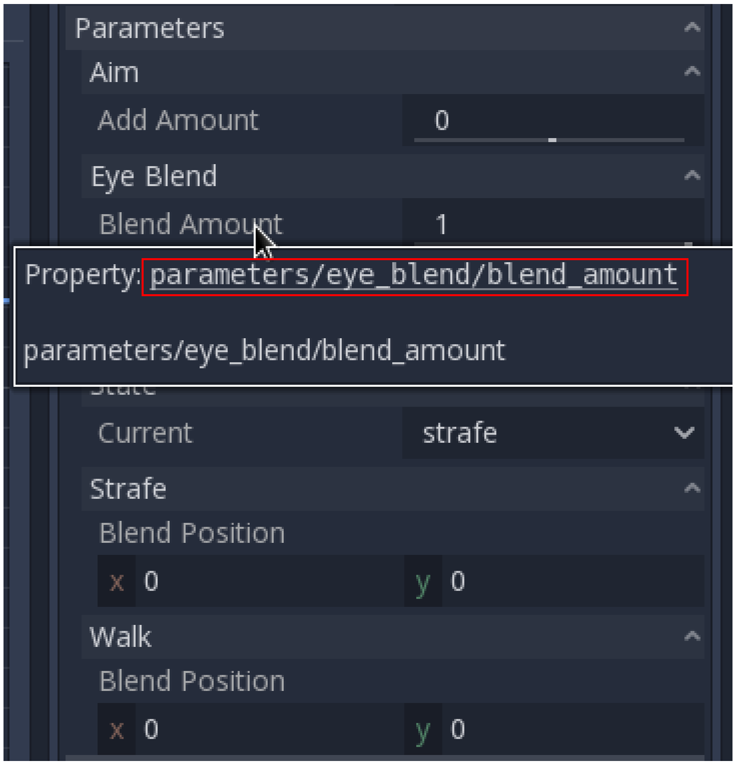
Blend Position: x 0 y 0

Node

Pause

This is handy because it makes it possible to animate them from an `AnimationPlayer`, or even the `AnimationTree` itself, allowing the realization of very complex animation logic.

To modify these values from code, the property path must be obtained. This is done easily by hovering the mouse over any of the parameters:


Parameters
Aim
Add Amount
0
Eye Blend
Blend Amount
1
Property: parameters/eye_blend/blend_amount

parameters/eye_blend/blend_amount

Current
strafe
Strafe
Blend Position
x 0
y 0
Walk
Blend Position
x 0
y 0

Which allows setting them or reading them:

GDScript

```
animation_tree.set("parameters/eye_blend/blend_amount", 1.0)
# Simpler alternative form:
animation_tree["parameters/eye_blend/blend_amount"] = 1.0
```

C#

```
animationTree.Set("parameters/eye_blend/blend_amount", 1.0);
```

State machine travel

One of the nice features in Godot's `StateMachine` implementation is the ability to travel. The graph can be instructed to go from the current state to another one, while visiting all the intermediate ones. This is done via the A* algorithm. If there is no path of transitions starting at the current state and finishing at the destination state, the graph teleports to the destination state.

To use the travel ability, you should first retrieve the AnimationNodeStateMachinePlayback object from the `AnimationTree` node (it is exported as a property).

GDScript

```
var state_machine = animation_tree["parameters/playback"]
```

C#

```
AnimationNodeStateMachinePlayback stateMachine =
(AnimationNodeStateMachinePlayback)animationTree.Get("parameter
s/playback");
```

Once retrieved, it can be used by calling one of the many functions it offers:

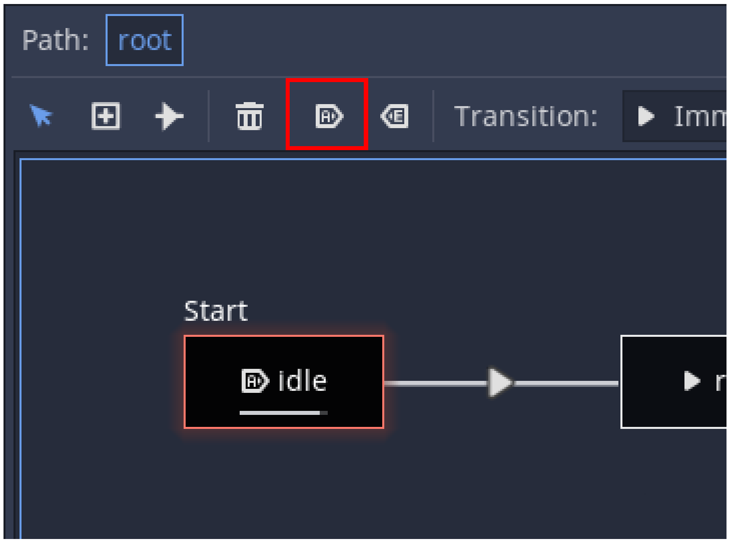
GDScript

```
state_machine.travel("SomeState")
```

C#

```
stateMachine.Travel("SomeState");
```

The state machine must be running before you can travel. Make sure to either call `start()` or choose a node to **Autoplay on Load**.



Path:
root
Transition:
Imm
Start
idle
r

Playing videos

Godot supports video playback with the VideoStreamPlayer node.

Supported playback formats

The only supported format in core is **Ogg Theora** (not to be confused with Ogg Vorbis audio). It's possible for extensions to bring support for additional formats, but no such extensions exist yet as of July 2022.

H.264 and H.265 cannot be supported in core Godot, as they are both encumbered by software patents. AV1 is royalty-free, but it remains slow to decode on the CPU and hardware decoding support isn't readily available on all GPUs in use yet.

WebM was supported in core in Godot 3.x, but support for it was removed in 4.0 as it was too buggy and difficult to maintain.

Note

You may find videos with an `.ogg` or `.ogx` extensions, which are generic extensions for data within an Ogg container.

Renaming these file extensions to `.ogv` *may* allow the videos to be imported in Godot. However, not all files with `.ogg` or `.ogx` extensions are videos - some of them may only contain audio.

Setting up VideoStreamPlayer

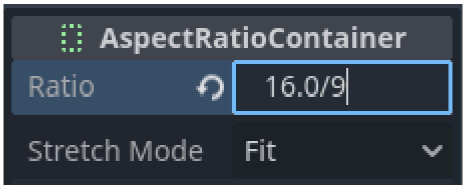
1. Create a VideoStreamPlayer node using the Create New Node dialog.
2. Select the VideoStreamPlayer node in the scene tree dock, go to the inspector and load an `.ogv` file in the Stream property.
    - If you don't have your video in Ogg Theora format yet, jump to Recommended Theora encoding settings.
3. If you want the video to play as soon as the scene is loaded, check **Autoplay** in the inspector. If not, leave **Autoplay** disabled and call `play()` on the VideoStreamPlayer node in a script to start playback when desired.

Handling resizing and different aspect ratios

By default in Godot 4.0, the VideoStreamPlayer will automatically be resized to match the video's resolution. You can make it follow usual Control sizing by enabling **Expand** on the VideoStreamPlayer node.

To adjust how the VideoStreamPlayer node resizes depending on window size, adjust the anchors using the **Layout** menu at the top of the 2D editor viewport. However, this setup may not be powerful enough to handle all use cases, such as playing fullscreen videos without distorting the video (but with empty space on the edges instead). For more control, you can use an AspectRatioContainer node, which is designed to handle this kind of use case:

Add an AspectRatioContainer node. Make sure it is not a child of any other container node. Select the AspectRatioContainer node, then set its **Layout** at the top of the 2D editor to **Full Rect**. Set **Ratio** in the AspectRatioContainer node to match your video's aspect ratio. You can use math formulas in the inspector to help yourself. Remember to make one of the operands a float. Otherwise, the division's result will always be an integer.


AspectRatioContainer

Ratio 16.0/9

Stretch Mode Fit

This will evaluate to (approximately) 1.777778

Once you've configured the AspectRatioContainer, reparent your VideoStreamPlayer node to be a child of the AspectRatioContainer node. Make sure **Expand** is enabled on the VideoStreamPlayer. Your video should now scale automatically to fit the whole screen while avoiding distortion.

See also

See Multiple resolutions for more tips on supporting multiple aspect ratios in your project.

Displaying a video on a 3D surface

Using a VideoStreamPlayer node as a child of a SubViewport node, it's possible to display any 2D node on a 3D surface. For example, this can be used to display animated billboards when frame-by-frame animation would require too much memory.

This can be done with the following steps:

1. Create a SubViewport node. Set its size to match your video's size in pixels.
2. Create a VideoStreamPlayer node *as a child of the SubViewport node* and specify a video path in it. Make sure **Expand** is disabled, and enable **Autoplay** if needed.
3. Create a MeshInstance3D node with a PlaneMesh or QuadMesh resource in its Mesh property. Resize the mesh to match the video's aspect ratio (otherwise, it will appear distorted).

4. Create a new StandardMaterial3D resource in the **Material Override** property in the GeometryInstance3D section.
5. Enable **Local To Scene** in the StandardMaterial3D's Resource section (at the bottom). This is *required* before you can use a ViewportTexture in its Albedo Texture property.
6. In the StandardMaterial3D, set the **Albedo > Texture** property to **New ViewportTexture**. Edit the new resource by clicking it, then specify the path to the SubViewport node in the **Viewport Path** property.
7. Enable **Albedo Texture Force sRGB** in the StandardMaterial3D to prevent colors from being washed out.
8. If the billboard is supposed to emit its own light, set **Shading Mode** to **Unshaded** to improve rendering performance.

See Using Viewports and the GUI in 3D demo [https://github.com/godotengine/godot-demo-projects/tree/master/viewport/gui_in_3d] for more information on setting this up.

Looping a video

For looping a video, the **Loop** property can be enabled. This will seamlessly restart the video when it reaches its end.

Note that setting the project setting **Video Delay Compensation** to a non-zero value might cause your loop to not be seamless, because the synchronization of audio and video takes place at the start of each loop causing occasional missed frames. Set **Video Delay Compensation** in your project settings to **0** to avoid frame drop issues.

Video decoding conditions and recommended resolutions

Video decoding is performed on the CPU, as GPUs don't have hardware acceleration for decoding Theora videos. Modern desktop CPUs can decode Ogg Theora videos at 1440p @ 60 FPS or more, but low-end mobile CPUs will likely struggle with high-resolution videos.

To ensure your videos decode smoothly on varied hardware:

- When developing games for desktop platforms, it's recommended to encode in 1080p at most (preferably at 30 FPS). Most people are still using 1080p or lower resolution displays, so encoding higher-resolution videos may not be worth the increased file size and CPU requirements.
- When developing games for mobile or web platforms, it's recommended to encode in 720p at most (preferably at 30 FPS or even lower). The visual difference between 720p and 1080p videos on a mobile device is usually not that noticeable.

Playback limitations

There are several limitations with the current implementation of video playback in Godot:

- Seeking a video to a certain point is not supported.
- Changing playback speed is not supported. VideoStreamPlayer also won't follow Engine.time_scale.
- Streaming a video from a URL is not supported.

Recommended Theora encoding settings

A word of advice is to **avoid relying on built-in Ogg Theora exporters** (most of the time). There are 2 reasons you may want to favor using an external program to encode your video:

- Some programs such as Blender can render to Ogg Theora. However, the default quality presets are usually very low by today's standards. You may be able to increase the quality options in the software you're using, but you may find the output quality to remain less than ideal (given the increased file size). This usually means that the software only supports encoding to constant bit rate (CBR), instead of variable bit rate (VBR). VBR encoding should be preferred in most scenarios as it provides a better quality to file size ratio.
- Some other programs can't render to Ogg Theora at all.

In this case, you can **render the video to an intermediate high-quality format** (such as a high-bitrate H.264 video) then re-encode it to Ogg Theora. Ideally, you should use a lossless or uncompressed format as an intermediate format to maximize the quality of the output Ogg Theora video, but this can require a lot of disk space.

HandBrake [https://handbrake.fr/] (GUI) and FFmpeg [https://ffmpeg.org/] (CLI) are popular open source tools for this purpose. FFmpeg has a steeper learning curve, but it's more powerful.

Here are example FFmpeg commands to convert an MP4 video to Ogg Theora. Since FFmpeg supports a lot of input formats, you should be able to use the commands below with almost any input video format (AVI, MOV, WebM, …).

Note

Make sure your copy of FFmpeg is compiled with libtheora and libvorbis support. You can check this by running `ffmpeg` without any arguments, then looking at the `configuration:` line in the command output.

Balancing quality and file size

The **video quality** level (`-q:v`) must be between `1` and `10`. Quality `6` is a good compromise between quality and file size. If encoding at a high resolution (such as 1440p or 4K), you will probably want to decrease `-q:v` to `5` to keep file sizes reasonable. Since pixel density is higher on a 1440p or 4K video, lower quality presets at higher resolutions will look as good or better compared to low-resolution videos.

The **audio quality** level (`-q:a`) must be between `-1` and `10`. Quality `6` provides a good compromise between quality and file size. In contrast to video quality, increasing audio quality doesn't increase the output file size nearly as much. Therefore, if you want the cleanest audio possible, you can increase this to `9` to get *perceptually lossless* audio. This is especially valuable if your input file already uses lossy audio compression. Higher quality audio does increase the CPU usage of the decoder, so it might lead to audio dropouts in case of high system load. See this page [https://wiki.hydrogenaud.io/index.php?title=Recommended_Ogg_Vorbis#Recommended_Encoder_Settings] for a table listing Ogg Vorbis audio quality presets and their respective variable bitrates.

FFmpeg: Convert while preserving original video resolution

The following command converts the video while keeping its original resolution. The video and audio's bitrate will be variable to maximize quality while saving space in parts of the video/audio that don't require a high bitrate (such as static scenes).

```
ffmpeg -i input.mp4 -q:v 6 -q:a 6 output.ogv
```

FFmpeg: Resize the video then convert it

The following command resizes a video to be 720 pixels tall (720p), while preserving its existing aspect ratio. This helps decrease the file size significantly if the source is recorded at a higher resolution than 720p:

```
ffmpeg -i input.mp4 -vf "scale=-1:720" -q:v 6 -q:a 6 output.ogv
```

Chroma Key Videos

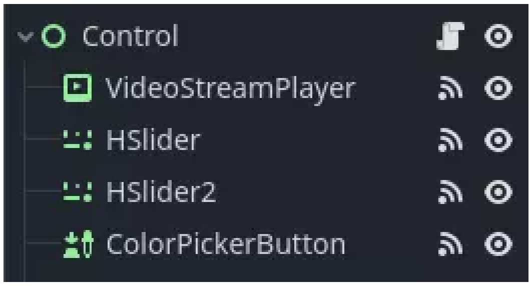
Chroma key, commonly known as the "green screen" or "blue screen" effect, allows you to remove a specific color from an image or video and replace it with another background. This effect is widely used in video production to composite different elements together seamlessly.

tutorials/animation/img/chroma_key_video.png

We will achieve the chroma key effect by writing a custom shader in GDScript and using a *VideoStreamPlayer* node to display the video content.

Scene Setup

Ensure that the scene contains a *VideoStreamPlayer* node to play the video and a *Control* node to hold the UI elements for controlling the chroma key effect.



Writing the Custom Shader

To implement the chroma key effect, follow these steps:

1. Select the *VideoStreamPlayer* node in the scene and go to its properties. Under *CanvasItem > Material*, create a new shader named "ChromaKeyShader.gdshader."
2. In the "ChromaKeyShader.gdshader" file, write the custom shader code as shown below:

```
shader_type canvas_item;

// Uniform variables for chroma key effect
uniform vec3 chroma_key_color : source_color = vec3(0.0, 1.0, 0.0);
uniform float pickup_range : hint_range(0.0, 1.0) = 0.1;
uniform float fade_amount : hint_range(0.0, 1.0) = 0.1;

void fragment() {
    // Get the color from the texture at the given UV coordinates
    vec4 color = texture(TEXTURE, UV);

    // Calculate the distance between the current color and the chroma key color
    float distance = length(color.rgb - chroma_key_color);

    // If the distance is within the pickup range, discard the pixel
    // the lesser the distance more likely the colors are
    if (distance <= pickup_range) {
        discard;
    }

    // Calculate the fade factor based on the pickup range and fade amount
    float fade_factor = smoothstep(pickup_range, pickup_range + fade_amount, distance);

    // Set the output color with the original RGB values and the calculated fade factor
    COLOR = vec4(color.rgb, fade_factor);
}
```

The shader uses the distance calculation to identify pixels close to the chroma key color and discards them, effectively removing the selected color. Pixels that are slightly further away from the chroma key color are faded based on the fade_factor, blending them smoothly with the surrounding colors. This process creates the desired chroma key effect, making it appear as if the background has been replaced with another image or video.

The code above represents a simple demonstration of the Chroma Key shader, and users can customize it according to their specific requirements.

UI Controls

To allow users to manipulate the chroma key effect in real-time, we created sliders in the *Control* node. The *Control* node's script contains the following functions:

GDScript

```
 extends Control

 func _on_color_picker_button_color_changed(color):
     # Update the "chroma_key_color" shader parameter of the
VideoStreamPlayer's material.

$VideoStreamPlayer.material.set("shader_parameter/chroma_key_co
lor", color)

 func _on_h_slider_value_changed(value):
     # Update the "pickup_range" shader parameter of the
VideoStreamPlayer's material.

$VideoStreamPlayer.material.set("shader_parameter/pickup_range"
, value)
```

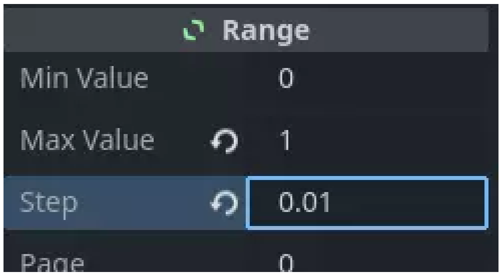
```
 func _on_h_slider_2_value_changed(value):
     # Update the "fade_amount" shader parameter of the
VideoStreamPlayer's material.

$VideoStreamPlayer.material.set("shader_parameter/fade_amount",
value)

func _on_video_stream_player_finished():
     # Restart the video playback when it's finished.
     $VideoStreamPlayer.play()
```

C#

```
using Godot;

public partial class MyControl : Control
{
    private VideoStreamPlayer _videoStreamPlayer;

    public override void _Ready()
    {
        _videoStreamPlayer = GetNode<VideoStreamPlayer>
("VideoStreamPlayer");
    }

    private void OnColorPickerButtonColorChanged(Color color)
    {
        // Update the "chroma_key_color" shader parameter of
the VideoStreamPlayer's material.

_videoStreamPlayer.Material.Set("shader_parameter/chroma_key_co
lor", color);
    }

    private void OnHSliderValueChanged(double value)
    {
        // Update the "pickup_range" shader parameter of the
VideoStreamPlayer's material.

_videoStreamPlayer.Material.Set("shader_parameter/pickup_range"
, value);
```

```
    }

    private void OnHSlider2ValueChanged(double value)
    {
        // Update the "fade_amount" shader parameter of the
VideoStreamPlayer's material.

_videoStreamPlayer.Material.Set("shader_parameter/fade_amount",
value);
    }

    private void OnVideoStreamPlayerFinished()
    {
        // Restart the video playback when it's finished.
        _videoStreamPlayer.Play();
    }
}
```

also make sure that the range of the sliders are appropriate, our settings are :


Range
Min Value 0
Max Value 1
Step 0.01
Page 0

Signal Handling

Connect the appropriate signal from the UI elements to the *Control* node's script. you created in the *Control* node's script to control the chroma key effect. These signal handlers will update the shader's uniform variables in response to user input.

Save and run the scene to see the chroma key effect in action! With the provided UI controls, you can now adjust the chroma key color, pickup range, and fade amount in real-time, achieving the desired chroma key functionality for your video content.

Creating movies

Godot can record **non-real-time** video and audio from any 2D or 3D project. This kind of recording is also called *offline rendering*. There are many scenarios where this is useful:

- Recording game trailers for promotional use.
- Recording cutscenes that will be displayed as pre-recorded videos in the final game. This allows for using higher quality settings (at the cost of file size), regardless of the player's hardware.
- Recording procedurally generated animations or motion design. User interaction remains possible during video recording, and audio can be included as well (although you won't be able to hear it while the video is recording).
- Comparing the visual output of graphics settings, shaders, or rendering techniques in an animated scene.

With Godot's animation features such as the AnimationPlayer node, Tweeners, particles and shaders, it can effectively be used to create any kind of 2D and 3D animations (and still images).

If you are already used to Godot's workflow, you may find yourself more productive by using Godot for video rendering compared to Blender. That said, renderers designed for non-real-time usage such as Cycles and Eevee can result in better visuals (at the cost of longer rendering times).

Compared to real-time video recording, some advantages of non-real-time recording include:

- Use any graphics settings (including extremely demanding settings) regardless of your hardware's capabilities. The output video will *always* have perfect frame pacing; it will never exhibit

dropped frames or stuttering. Faster hardware will allow you to render a given animation in less time, but the visual output remains identical.
- Render at a higher resolution than the screen resolution, without having to rely on driver-specific tools such as NVIDIA's Dynamic Super Resolution or AMD's Virtual Super Resolution.
- Render at a higher framerate than the video's target framerate, then post-process to generate high-quality motion blur. This also makes effects that converge over several frames (such as temporal antialiasing, SDFGI and volumetric fog) look better.

Warning

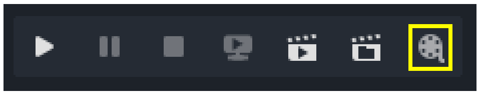
This feature is not designed for capturing real-time footage during gameplay.

Players should use something like OBS Studio [https://obsproject.com/] or SimpleScreenRecorder [https://www.maartenbaert.be/simplescreenrecorder/] to record gameplay videos, as they do a much better job at intercepting the compositor than Godot can do using Vulkan or OpenGL natively.

That said, if your game runs at near-real-time speeds when capturing, you can still use this feature (but it will lack audible sound playback, as sound is saved directly to the video file).

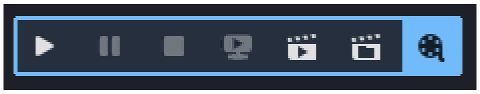
Enabling Movie Maker mode

To enable Movie Maker mode, click the "movie reel" button in the top-right corner of the editor *before* running the project:



Movie Maker mode is disabled, click the "movie reel" icon to enable

The icon gets a background matching the accent color when Movie Maker mode is enabled:



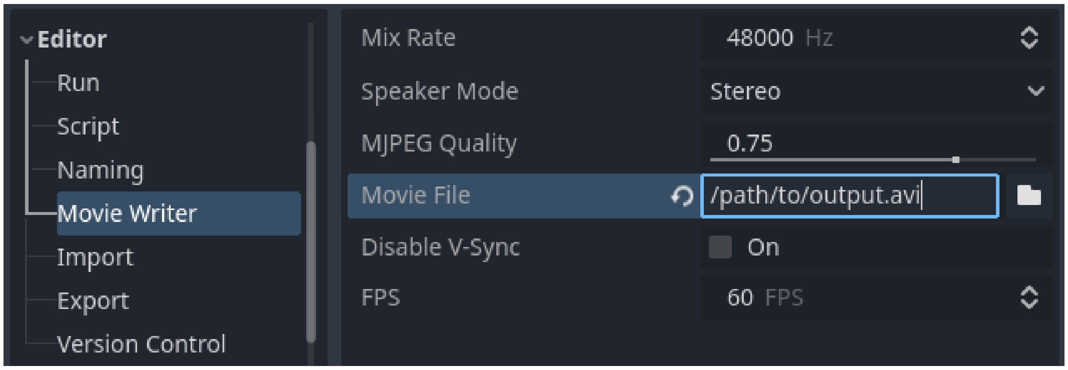
Movie Maker mode is enabled, click the "movie reel" icon again to disable

Movie Maker status is **not** persisted when the editor quits, so you must re-enable Movie Maker mode again after restarting the editor if needed.

Note

Toggling Movie Maker mode while running the project will not have any effect until the project is restarted.

Before you can record video by running the project, you still need to configure the output file path. This path can be set for all scenes in the Project Settings:



Mix Rate 48000 Hz

Speaker Mode Stereo

MJPEG Quality 0.75

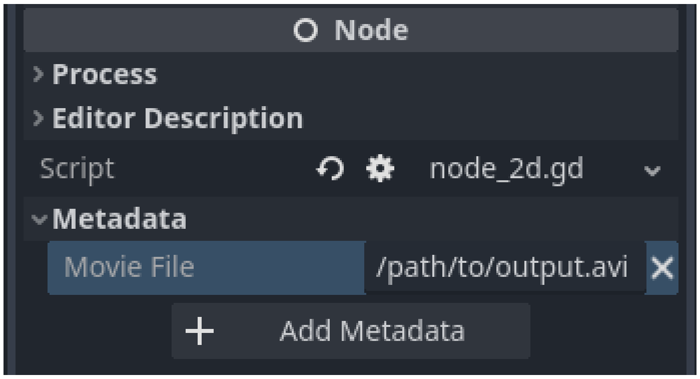
Movie File /path/to/output.avi

Disable V-Sync ☐ On

FPS 60 FPS

Movie Maker project settings (with Advanced toggle enabled)

Alternatively, you can set the output file path on a per-scene basis by adding a String metadata with the name `movie_file` to the scene's **root node**. This is only used when the main scene is set to the scene in question, or when running the scene directly by pressing `F6` (`Cmd + R` on macOS).



Script node_2d.gd

Movie File /path/to/output.avi

Inspector view after creating a `movie_file` metadata of type String

The path specified in the project settings or metadata can be either absolute, or relative to the project root.

Once you've configured and enabled Movie Maker mode, it will be automatically used when running the project from the editor.

Command line usage

Movie Maker can also be enabled from the command line:

```
godot --path /path/to/your_project --write-movie output.avi
```

If the output path is relative, then it is **relative to the project folder**, not the current working directory. In the above example, the file will be written to `/path/to/your_project/output.avi`. This behavior is similar to the `--export-release` command line argument.

Since Movie Maker's output resolution is set by the viewport size, you can adjust the window size on startup to override it if the project uses the `disabled` or `canvas_items` stretch mode:

```
godot --path /path/to/your_project --write-movie output.avi --
resolution 1280x720
```

Note that the window size is clamped by your display's resolution. See Rendering at a higher resolution than the screen resolution if you need to record a video at a higher resolution than the screen resolution.

The recording FPS can also be overridden on the command line, without having to edit the Project Settings:

```
godot --path /path/to/your_project --write-movie output.avi --
fixed-fps 30
```

Note

The `--write-movie` and `--fixed-fps` command line arguments are both available in exported projects. Movie Maker mode cannot be toggled while the project is running, but you can use the OS.execute() method to run a second instance of the exported project that will record a video file.

Choosing an output format

Output formats are provided by the MovieWriter class. Godot has 2 built-in MovieWriters, and more can be implemented by extensions:

AVI (recommended)

AVI container with MJPEG for video and uncompressed audio. Features lossy video compression, resulting in medium file sizes and fast encoding. The lossy compression quality can be adjusted by changing **Editor > Movie Writer > MJPEG Quality**.

The resulting file can be viewed in most video players, but it must be converted to another format for viewing on the web or by Godot with the VideoStreamPlayer node. MJPEG does not support transparency. AVI output is currently limited to a file of 4 GB in size at most.

To use AVI, specify a path to an `.avi` file to be created in the **Editor > Movie Writer > Movie File** project setting.

PNG

PNG image sequence for video and WAV for audio. Features lossless video compression, at the cost of large file sizes and slow encoding. This is designed to be encoded to a video file with an external tool after recording.

Transparency is supported, but the root viewport **must** have its `transparent_bg` property set to `true` for transparency to be visible on the output image. This can be achieved by enabling the **Rendering > Transparent Background** advanced project setting. **Display > Window > Size > Transparent** and **Display > Window > Per Pixel Transparency > Enabled** can optionally be enabled to allow transparency to be previewed while recording the video, but they do not have to be enabled for the output image to contain transparency.

To use PNG, specify a `.png` file to be created in the **Editor > Movie Writer > Movie File** project setting. The generated `.wav` file will have the same name as the `.png` file (minus the extension).

Custom

If you need to encode directly to a different format or pipe a stream through third-party software, you can extend the MovieWriter class to create your own movie writers. This should typically be done using GDExtension for performance reasons.

Configuration

In the **Editor > Movie Writer** section of the Project Settings, there are several options you can configure. Some of them are only visible after enabling the **Advanced** toggle in the top-right corner of the Project Settings dialog.

- **Mix Rate Hz:** The audio mix rate to use in the recorded audio when writing a movie. This can be different from the project's mix rate, but this value must be divisible by the recorded FPS to prevent audio from desynchronizing over time.
- **Speaker Mode:** The speaker mode to use in the recorded audio when writing a movie (stereo, 5.1 surround or 7.1 surround).
- **MJPEG Quality:** The JPEG quality to use when writing a video to an AVI file, between `0.01` and `1.0` (inclusive). Higher quality values result in better-looking output at the cost of larger file sizes. Recommended quality values are between `0.75` and `0.9`. Even at quality `1.0`, JPEG compression remains lossy. This setting does not affect audio quality and is ignored when writing to a PNG image sequence.
- **Movie File:** The output path for the movie. This can be absolute or relative to the project root.
- **Disable V-Sync:** If enabled, requests V-Sync to be disabled when writing a movie. This can speed up video writing if the hardware is fast enough to render, encode and save the video at a framerate higher than the monitor's refresh rate. This setting

has no effect if the operating system or graphics driver forces V-Sync with no way for applications to disable it.

- **FPS:** The rendered frames per second in the output movie. Higher values result in smoother animation, at the cost of longer rendering times and larger output file sizes. Most video hosting platforms do not support FPS values higher than 60, but you can use a higher value and use that to generate motion blur.

Note

When using the `disabled` or `2d` stretch modes, the output file's resolution is set by the window size. Make sure to resize the window *before* the splash screen has ended. For this purpose, it's recommended to adjust the **Display > Window > Size > Window Width Override** and **Window Height Override** advanced project settings.

See also Rendering at a higher resolution than the screen resolution.

Quitting Movie Maker mode

To safely quit a project that is using Movie Maker mode, use the X button at the top of the window, or call `get_tree().quit()` in a script. You can also use the `--quit-after N` command line argument where `N` is the number of frames to render before quitting.

Pressing `F8` (`Cmd + .` on macOS) or pressing `Ctrl + C` on the terminal running Godot is **not recommended**, as it will result in an improperly formatted AVI file with no duration information. For PNG image sequences, PNG images will not be negatively altered, but the associated WAV file will still lack duration information.

Some video players may still be able to play the AVI or WAV file with working video and audio. However, software that makes use of the AVI or WAV file such as video editors may not be able to open the file. Using a video converter program can help in those cases.

If you're using an AnimationPlayer to control a "main action" in the scene (such as camera movement), you can enable the **Movie Quit On Finish** property on the AnimationPlayer node in question. When enabled, this property will make Godot quit on its own when an animation is done playing *and* the engine is running in Movie Maker mode. Note that *this property has no effect on looping animations*. Therefore, you need to make sure that the animation is set as non-looping.

Using high-quality graphics settings

The `movie` feature tag can be used to override specific project settings. This is useful to enable high-quality graphics settings that wouldn't be fast enough to run in real-time speeds on your hardware. Remember that putting every setting to its maximum value can still slow down movie saving speed, especially when recording at higher resolutions. Therefore, it's still recommended to only increase graphics settings if they make a meaningful difference in the output image.

This feature tag can also be queried in a script to increase quality settings that are set in the Environment resource. For example, to further improve SDFGI detail and reduce light leaking:

GDScript

```
extends Node3D

func _ready():
    if OS.has_feature("movie"):
        # When recording a movie, improve SDFGI cell density
        # without decreasing its maximum distance.
        get_viewport().world_3d.environment.sdfgi_min_cell_size
*= 0.25
        get_viewport().world_3d.environment.sdfgi_cascades = 8
```

C#

```
using Godot;

public partial class MyNode3D : Node3D
{
    public override void _Ready()
    {
        if (OS.HasFeature("movie"))
        {
            // When recording a movie, improve SDFGI cell
density
            // without decreasing its maximum distance.
            GetViewport().World3D.Environment.SdfgiMinCellSize
*= 0.25f;
            GetViewport().World3D.Environment.SdfgiCascades =
8;
        }
    }
}
```

Rendering at a higher resolution than the screen resolution

The overall rendering quality can be improved significantly by rendering at high resolutions such as 4K or 8K.

Note

For 3D rendering, Godot provides a **Rendering > Scaling 3D > Scale** advanced project setting, which can be set above `1.0` to obtain *supersample antialiasing*. The 3D rendering is then *downsampled* when it's drawn on the viewport. This provides an expensive but high-quality form of antialiasing, without increasing the final output resolution.

Consider using this project setting first, as it avoids slowing down movie writing speeds and increasing output file size compared to actually increasing the output resolution.

If you wish to render 2D at a higher resolution, or if you actually need the higher raw pixel output for 3D rendering, you can increase the resolution above what the screen allows.

By default, Godot uses the `disabled` stretch modes in projects. If using `disabled` or `canvas_items` stretch mode, the window size dictates the output video resolution.

On the other hand, if the project is configured to use the `viewport` stretch mode, the viewport resolution dictates the output video resolution. The viewport resolution is set using the **Display > Window > Size > Viewport Width** and **Viewport Height** project settings. This can be used to render a video at a higher resolution than the screen resolution.

To make the window smaller during recording without affecting the output video resolution, you can set the **Display > Window > Size > Window Width Override** and **Window Height Override** advanced project settings to values greater than `0`.

To apply a resolution override only when recording a movie, you can override those settings with the `movie` feature tag.

Post-processing steps

Some common post-processing steps are listed below.

Note

When using several post-processing steps, try to perform all of them in a single FFmpeg command. This will save encoding time and improve quality by avoiding multiple lossy encoding steps.

Converting AVI video to MP4

While some platforms such as YouTube support uploading the AVI file directly, many others will require a conversion step beforehand. HandBrake [https://handbrake.fr/] (GUI) and FFmpeg [https://ffmpeg.org/] (CLI) are popular open source tools for this purpose. FFmpeg has a steeper learning curve, but it's more powerful.

The command below converts an AVI video to an MP4 (H.264) video with a Constant Rate Factor (CRF) of 15. This results in a relatively large file, but is well-suited for platforms that will re-encode your videos to reduce their size (such as most video sharing websites):

```
ffmpeg -i input.avi -crf 15 output.mp4
```

To get a smaller file at the cost of quality, *increase* the CRF value in the above command.

To get a file with a better size/quality ratio (at the cost of slower encoding times), add `-preset veryslow` before `-crf 15` in the above command. On the contrary, `-preset veryfast` can be used to achieve faster encoding at the cost of a worse size/quality ratio.

Converting PNG image sequence + WAV audio to a video

If you chose to record a PNG image sequence with a WAV file beside it, you need to convert it to a video before you can use it elsewhere.

The filename for the PNG image sequence generated by Godot always contains 8 digits, starting at 0 with zero-padded numbers. If you specify an output path `folder/example.png`, Godot will write `folder/example00000000.png`, `folder/example00000001.png`, and so on in that folder. The audio will be saved at `folder/example.wav`.

The FPS is specified using the `-r` argument. It should match the FPS specified during recording. Otherwise, the video will appear to be slowed down or sped up, and audio will be out of sync with the video.

```
ffmpeg -r 60 -i input%08d.png -i input.wav -crf 15 output.mp4
```

If you recorded a PNG image sequence with transparency enabled, you need to use a video format that supports storing transparency. MP4/H.264 doesn't support storing transparency, so you can use WebM/VP9 as an alternative:

```
ffmpeg -r 60 -i input%08d.png -i input.wav -c:v libvpx-vp9 -crf
15 -pix_fmt yuva420p output.webm
```

Cutting video

You can trim parts of the video you don't want to keep after the video is recorded. For example, to discard everything before 12.1 seconds and keep only 5.2 seconds of video after that point:

```
ffmpeg -i input.avi -ss 00:00:12.10 -t 00:00:05.20 -crf 15
output.mp4
```

Cutting videos can also be done with the GUI tool LosslessCut [https://mifi.github.io/lossless-cut/].

Resizing video

The following command resizes a video to be 1080 pixels tall (1080p), while preserving its existing aspect ratio:

```
ffmpeg -i input.avi -vf "scale=-1:1080" -crf 15 output.mp4
```

Reducing framerate

The following command changes a video's framerate to 30 FPS, dropping some of the original frames if there are more in the input video:

```
ffmpeg -i input.avi -r 30 -crf 15 output.mp4
```

Generating accumulation motion blur with FFmpeg

Godot does not have built-in support for motion blur, but it can still be created in recorded videos.

If you record the video at a multiple of the original framerate, you can blend the frames together then reduce the frameate to produce a video with *accumulation motion blur*. This motion blur can look very good, but it can take a long time to generate since you have to render many more frames per second (on top of the time spent on post-processing).

Example with a 240 FPS source video, generating 4× motion blur and decreasing its output framerate to 60 FPS:

```
ffmpeg -i input.avi -vf "tmix=frames=4, fps=60" -crf 15
output.mp4
```

This also makes effects that converge over several frames (such as temporal antialiasing, SDFGI and volumetric fog) converge faster and therefore look better, since they'll be able to work with more data at a given time. See Reducing framerate if you want to get this benefit without adding motion blur.

Assets pipeline

- Import process
- Importing images
- Importing audio samples
- Importing translations
- Importing 3D scenes
- Retargeting 3D Skeletons
- Exporting 3D scenes
- Blender ESCN exporter

Import process

Importing assets in Godot

To import assets in Godot, place your assets (image files, scenes, audio files, fonts, etc) directly in the project folder. There are 2 ways to achieve this:

- **For any file type:** Copy files manually with your operating system's file manager.
- **For file types that can be imported by Godot:** Drag-and-drop files from the operating system's file manager to the editor's FileSystem dock. This only works with *resource* file types (i.e. file types that Godot can import).

Godot will automatically import these files internally and keep the imported resources hidden in a `res://.godot/imported/` folder.

This means that when trying to access imported assets through code, you need to use the Resource Loader as it will automatically take into account where the internal files are saved. If you try and access an imported asset using the FileAccess class, it will work in the editor, but **it will break in the exported project**.

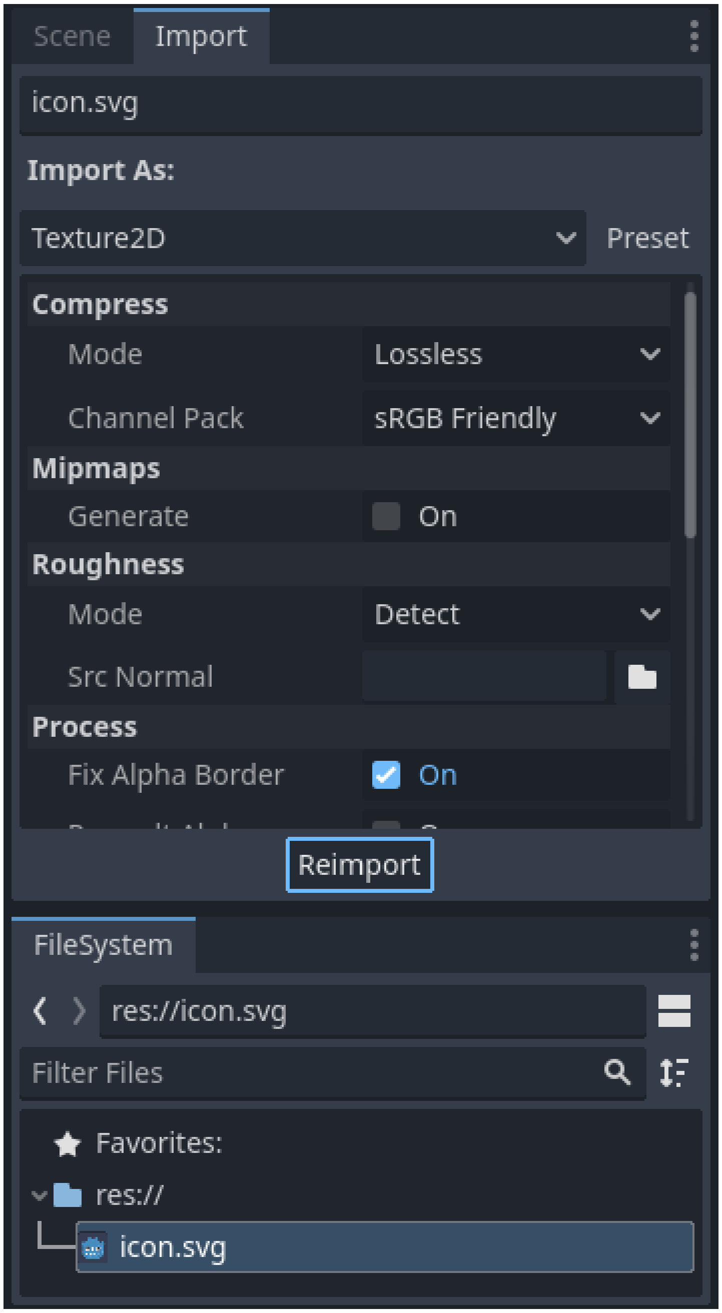
However, the Resource Loader cannot access non-imported files. Only the FileAccess class can.

Changing import parameters

Note

Import parameters are only present in *non-native* Godot resource types. This means Godot's own scene and resource file formats (`.tscn`, `.scn`, `.tres`, `.res`) don't have import options you can select in the Import dock.

To change the import parameters of an asset in Godot, select the relevant resource in the FileSystem dock:


Scene
Import
Preset
Reimport
FileSystem
Filter Files
Favorites:

icon.svg

Import As: Texture2D

Compress
- Mode: Lossless
- Channel Pack: sRGB Friendly

Mipmaps
- Generate: ☐ On

Roughness
- Mode: Detect
- Src Normal:

Process
- Fix Alpha Border: ☑ On

res://icon.svg

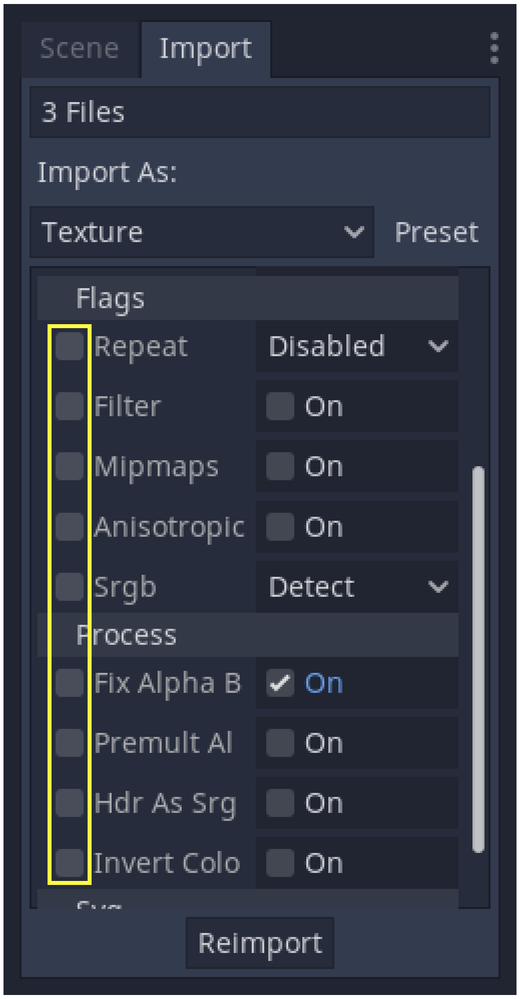
res://
- icon.svg

After adjusting the parameters, click **Reimport**. Be careful: if you select another file in the FileSystem dock before clicking **Reimport**, changes will be discarded. After clicking **Reimport**, the chosen parameters will only be used for this asset and on future reimports.

Changing the import parameters of several assets at the same time is also possible. Simply select all of them together in the resources dock and the exposed parameters will apply to all of them when reimporting.

Reimporting multiple assets

While working on a project you may find that several assets need to have the same parameters changed, such as enabling mipmaps, but you only want those specific parameters changed. To do this, select every asset you want to reimport in the file system. In the import tab there will now be a checkbox to the left of every import parameter.



3 Files

Import As: Texture

Flags

☐ Repeat Disabled

☐ Filter ☐ On

☐ Mipmaps ☐ On

☐ Anisotropic ☐ On

☐ Srgb Detect

Process

☐ Fix Alpha B ☑ On

☐ Premult Al ☐ On

☐ Hdr As Srg ☐ On

☐ Invert Colo ☐ On

Select the checkbox of the parameters you want to change on your imported assets, then change the parameters normally. Finally, click the reimport button and every selected asset will be reimported with only those parameters changed.

Automatic reimport

When the MD5 checksum of the source asset changes, Godot will perform an automatic reimport of it, applying the preset configured for that specific asset.

Files generated

Importing will add an extra `<asset>.import` file next to the source file, containing the import configuration.

Make sure to commit these files to your version control system, as these files contain important metadata.

```
ls
example.png
example.png.import
project.godot
```

Additionally, extra assets will be present in the hidden `res://.godot/imported/` folder:

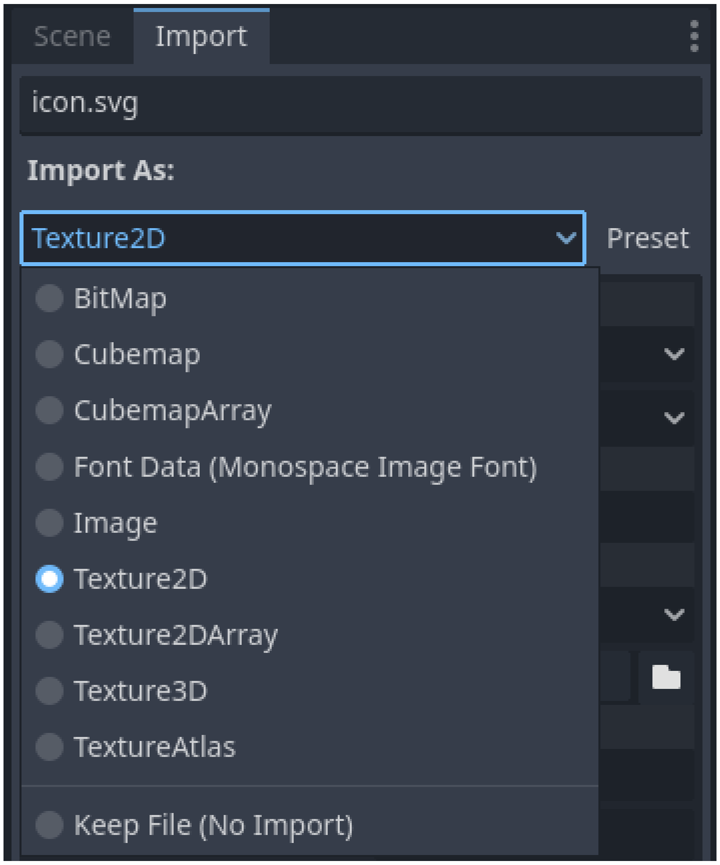
```
ls .godot/imported
example.png-218a8f2b3041327d8a5756f3a245f83b.ctex
example.png-218a8f2b3041327d8a5756f3a245f83b.md5
```

If any of the files present in this folder is erased (or the whole folder), the asset or assets will be reimported automatically. As such, committing the `.godot/` folder to the version control system is not recommended. While committing this folder can shorten reimporting time when checking out on another computer, it requires considerably more space and bandwidth.

The default version control metadata that can be generated on project creation will automatically ignore the `.godot/` folder.

Changing import resource type

Some source assets can be imported as different types of resources. For this, select the relevant type of resource desired then click **Reimport**:


icon.svg

Import As: Texture2D

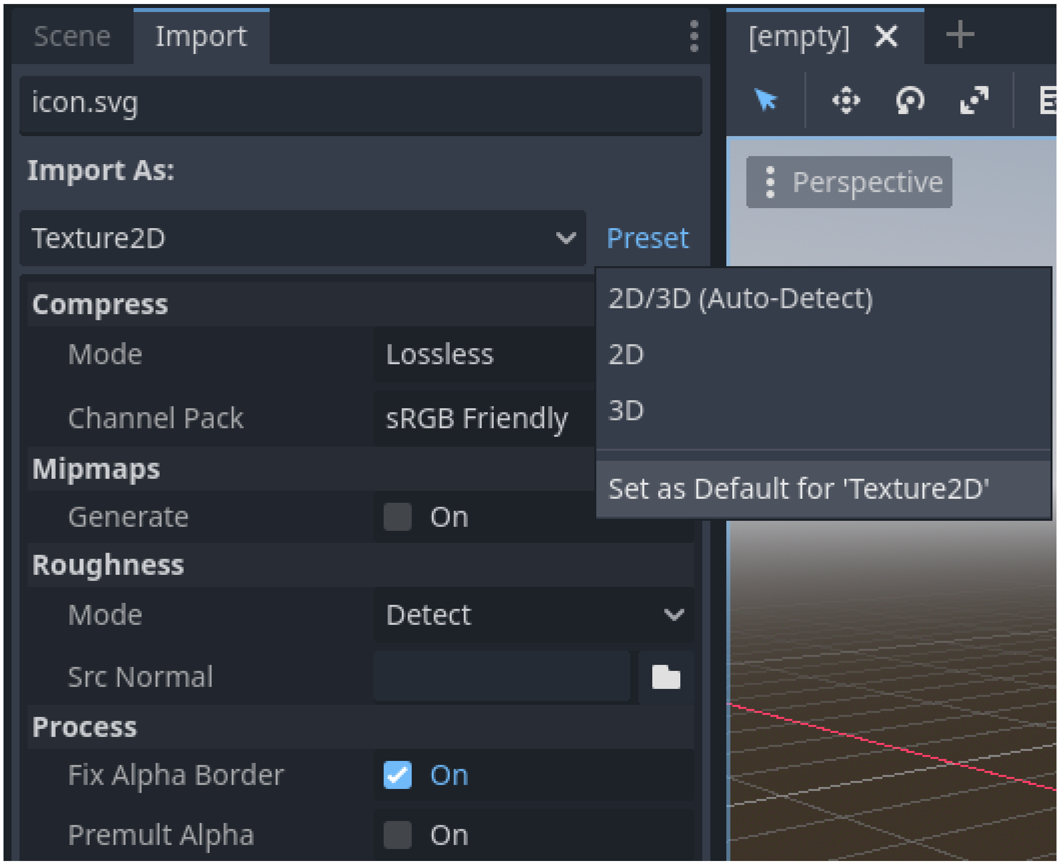
Scene
Import
Preset
○ BitMap
○ Cubemap
○ CubemapArray
○ Font Data (Monospace Image Font)
○ Image
● Texture2D
○ Texture2DArray
○ Texture3D
○ TextureAtlas
○ Keep File (No Import)

Select `Keep File (exported as is)` as resource type to skip file import, files with this resource type will be preserved as is during project export.

Select `Skip File (not exported)` as resource type to skip file import and ignore file during project export.

Changing default import parameters

Different types of projects might require different defaults. Changing the import options to a predefined set of options can be achieved by using the **Preset...** Menu. Besides some resource types offering presets, the default settings can be saved and cleared too:



icon.svg

Import As:

Texture2D

Compress

Mode: Lossless

Channel Pack: sRGB Friendly

Mipmaps

Generate: ☐ On

Roughness

Mode: Detect

Src Normal:

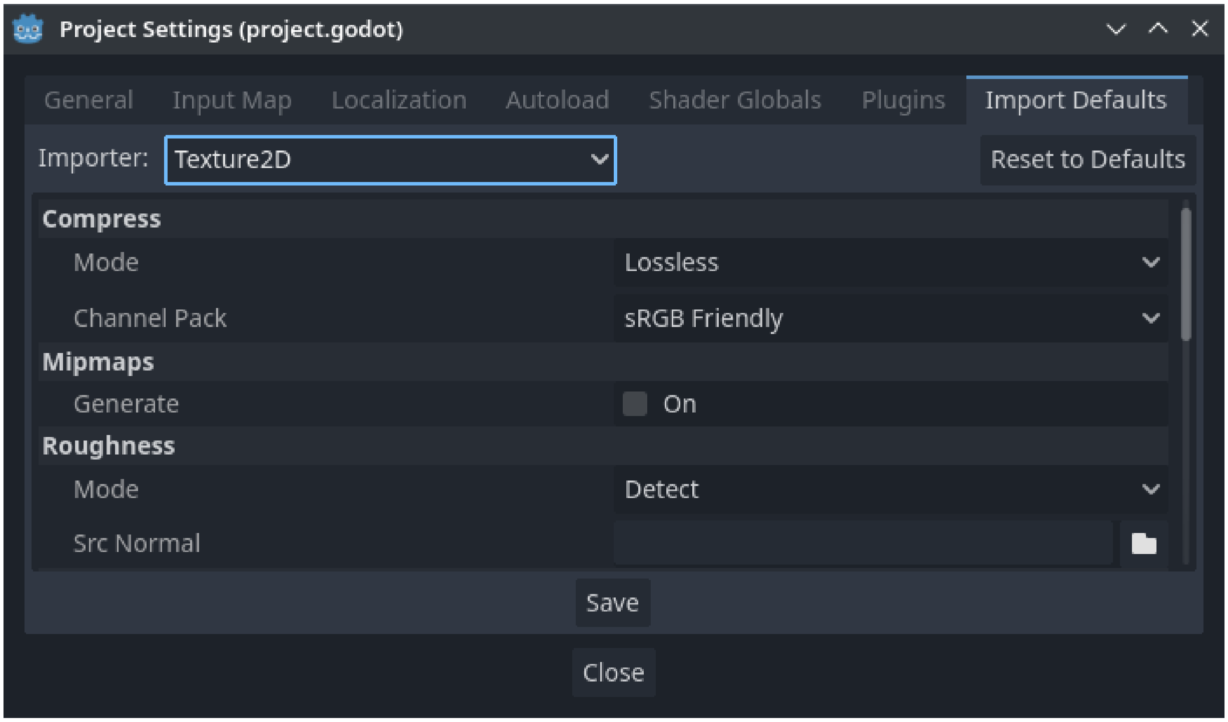
Process

Fix Alpha Border: ☑ On

Premult Alpha: ☐ On

Scene
Import
[empty]
Perspective
Preset
2D/3D (Auto-Detect)
2D
3D
Set as Default for 'Texture2D'

The default import parameters for a given resource type can be changed project-wide using the **Import Defaults** tab of the Project Settings dialog:


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Reset to Defaults
Save
Close

Importer: Texture2D

Compress

Mode: Lossless

Channel Pack: sRGB Friendly

Mipmaps

Generate: ☐ On

Roughness

Mode: Detect

Src Normal:

Further reading

This workflow takes a little time to get used to, but it enforces a more correct way to deal with resources.

There are many types of assets available for import. Continue reading to understand how to work with all of them:

- Importing images
- Importing audio samples
- Importing 3D scenes
- Importing translations

Importing images

Supported image formats

Godot can import the following image formats:

- BMP (`.bmp`) - No support for 16-bit per pixel images. Only 1-bit, 4-bit, 8-bit, 24-bit, and 32-bit per pixel images are supported.
- DirectDraw Surface (`.dds`) - If mipmaps are present in the texture, they will be loaded directly. This can be used to achieve effects using custom mipmaps.
- Khronos Texture (`.ktx`) - Decoding is done using libktx [https://github.com/KhronosGroup/KTX-Software]. Only supports 2D images. Cubemaps, texture arrays and de-padding are not supported.
- OpenEXR (`.exr`) - Supports HDR (highly recommended for panorama skies).
- Radiance HDR (`.hdr`) - Supports HDR (highly recommended for panorama skies).
- JPEG (`.jpg`, `.jpeg`) - Doesn't support transparency per the format's limitations.
- PNG (`.png`) - Precision is limited to 8 bits per channel upon importing (no HDR images).
- Truevision Targa (`.tga`)
- SVG (`.svg`) - SVGs are rasterized using ThorVG [https://www.thorvg.org/] when importing them. Support is limited [https://www.thorvg.org/about#:~:text=among%20the%20svg%20tiny%20specs%2C%20yet%20unsupported%20features%20in%20the%20thorvg%20are%20the%20following]; complex vectors may not render correctly. Text must be converted to paths; otherwise, it won't appear in the rasterized image. You can check whether ThorVG can render a certain vector correctly using its web-based viewer [https://www.thorvg.org/viewer]. For complex vectors, rendering them to PNGs using Inkscape [https://inkscape.org/] is often a better solution. This can be automated thanks to its command-line interface [https://wiki.inkscape.org/wiki/index.php/Using_the_Command_Line#Export_files].
- WebP (`.webp`) - WebP files support transparency and can be compressed lossily or losslessly. The precision is limited to 8 bits per channel.

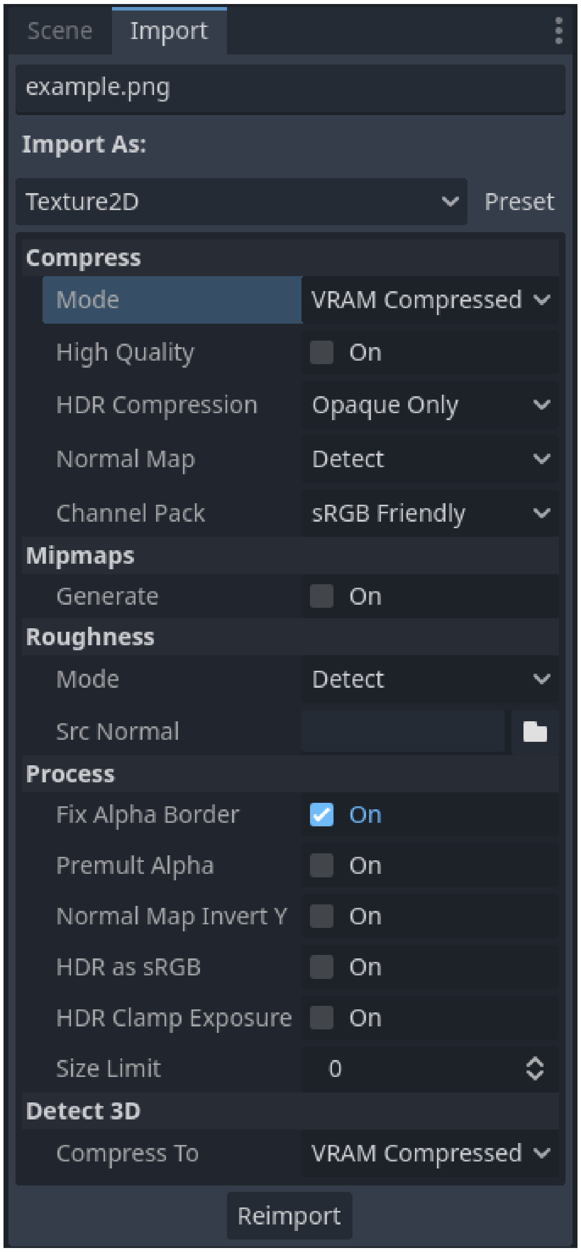
Note

If you've compiled the Godot editor from source with specific modules disabled, some formats may not be available.

Importing textures

The default action in Godot is to import images as textures. Textures are stored in video memory. Their pixel data can't be accessed directly from the CPU without converting them back to an Image in a script. This is what makes drawing them efficient.

There are over a dozen import options that can be adjusted after selecting an image in the FileSystem dock:



example.png

Import As: Texture2D

Compress

Mode: VRAM Compressed
High Quality: ☐ On
HDR Compression: Opaque Only
Normal Map: Detect
Channel Pack: sRGB Friendly

Mipmaps

Generate: ☐ On

Roughness

Mode: Detect
Src Normal:

Process

Fix Alpha Border: ☑ On
Premult Alpha: ☐ On
Normal Map Invert Y: ☐ On
HDR as sRGB: ☐ On
HDR Clamp Exposure: ☐ On
Size Limit: 0

Detect 3D

Compress To: VRAM Compressed

Import options in the Import dock after selecting an image in the FileSystem dock. Some of these options are only visible with certain compression modes.

Changing import type

It is possible to choose other types of imported resources in the Import dock:

- **BitMap:** 1-bit monochrome texture (intended to be used as a click mask in TextureButton and TouchScreenButton). This resource type cannot be displayed directly onto 2D or 3D nodes, but the pixel values can be queried from a script using get_bit.

- **Cubemap:** Import the texture as a 6-sided cubemap, with interpolation between the cubemap's sides (seamless cubemaps), which can be sampled in custom shaders.
- **CubemapArray:** Import the texture as a collection of 6-sided cubemaps, which can be sampled in custom shaders. This resource type can only be displayed when using the Forward+ or Forward Mobile rendering methods, not Compatibility.
- **Font Data (Monospace Image Font):** Import the image as a bitmap font where all characters have the same width. See Using Fonts.
- **Image:** Import the image as-is. This resource type cannot be displayed directly onto 2D or 3D nodes, but the pixel values can be queried from a script using get_pixel.
- **Texture2D:** Import the image as a 2-dimensional texture, suited for display on 2D and 3D surfaces. This is the default import mode.
- **Texture2DArray:** Import the image as a collection of 2-dimensional textures. Texture2DArray is similar to a 3-dimensional texture, but without interpolation between layers. Built-in 2D and 3D shaders cannot display texture arrays, so you must create a custom shader in 2D or 3D to display a texture from a texture array.
- **Texture3D:** Import the image as a 3-dimensional texture. This is *not* a 2D texture applied onto a 3D surface. Texture3D is similar to a texture array, but with interpolation between layers. Texture3D is typically used for FogMaterial density maps in volumetric fog, particle attractor vector fields, Environment 3D LUT color correction, and custom shaders.
- **TextureAtlas:** Import the image as an *atlas* of different textures. Can be used to reduce memory usage for animated 2D sprites. Only supported in 2D due to missing support in built-in 3D shaders.

For **Cubemap**, the expected image order is X+, X-, Y+, Y-, Z+, Z- (in Godot's coordinate system, so Y+ is "up" and Z- is "forward"). Here are templates you can use for cubemap images (right-click > **Save Link As…**):

- `2×3 cubemap template (default layout option)`
- `3×2 cubemap template`
- `1×6 cubemap template`
- `6×1 cubemap template`

Detect 3D

The default import options (no mipmaps and **Lossless** compression) are suited for 2D, but are not ideal for most 3D projects. **Detect 3D** makes Godot aware of when a texture is used in a 3D scene (such as a texture in a BaseMaterial3D). If this happens, several import options are changed so the texture flags are friendlier to 3D. Mipmaps are enabled and the compression mode is changed to **VRAM Compressed** unless Detect 3D > Compress To is changed. The texture is also reimported automatically.

A message is printed to the Output panel when a texture is detected to be used in 3D.

If you run into quality issues when a texture is detected to be used in 3D (e.g. for pixel art textures), change the Detect 3D > Compress To option before using the texture in 3D, or change Compress > Mode to **Lossless** after using the texture in 3D. This is preferable to disabling **Detect 3D**, as mipmap generation remains enabled to prevent textures from looking grainy at a distance.

Import options

See also

In Godot 4.0, changing the texture filter and repeat mode is no longer done in the import options.

Instead, texture filter and repeat modes are changed in the CanvasItem properties in 2D (with a project setting acting as a default), and in a per-material configuration in 3D. In custom shaders, filter and repeat mode is changed on the `sampler2D` uniform using hints described in the Shading language documentation.

Compress > Mode

Images are one of the largest assets in a game. To handle them efficiently, they need to be compressed. Godot offers several compression methods, depending on the use case.

- **Lossless:** This is the default and most common compression mode for 2D assets. It shows assets without any kind of artifacting, and disk compression is decent. It will use considerably more amount of video memory than VRAM Compression, though. This is also the recommended setting for pixel art.

- **Lossy:** This is a good choice for large 2D assets. It has some artifacts, but less than VRAM compression and the file size is several times lower compared to Lossless or VRAM Uncompressed. Video memory usage isn't decreased by this mode; it's the same as with Lossless or VRAM Uncompressed.
- **VRAM Compressed:** This is the default and most common compression mode for 3D assets. Size on disk is reduced and video memory usage is also decreased considerably (usually by a factor between 4 and 6). This mode should be avoided for 2D as it exhibits noticeable artifacts, especially for lower-resolution textures.
- **VRAM Uncompressed:** Only useful for formats that can't be compressed, such as raw floating-point images.
- **Basis Universal:** This alternative VRAM compression mode encodes the texture to a format that can be transcoded to most GPU-compressed formats at load-time. This provides very small files that make use of VRAM compression, at the cost of lower quality compared to VRAM Compressed and slow compression times. VRAM usage is usually the same as VRAM Compressed. Basis Universal does not support floating-point image formats (the engine will internally fall back to VRAM Compressed instead).

Note

Even in 3D, "pixel art" textures should have VRAM compression disabled as it will negatively affect their appearance, without improving performance significantly due to their low resolution.

In this table, each of the 5 options are described together with their advantages and disadvantages (🟢 = best, 🔴 = worst):

Compress mode	Lossless	Lossy	VRAM Compressed	VRAM Uncompressed	Basis Universal
Description	Stored as Lossless WebP / PNG	Stored as Lossy WebP	Stored as S3TC, BPTC or ETC2 depending on platform	Stored as raw pixels	Transcoded to VRAM Compressed format

Compress mode	Lossless	Lossy	VRAM Compressed	VRAM Uncompressed	Basis Universal
Size on disk	🟡 Small	🟢 Very small	🟡 Small	🔴 Large	🟢 Very small
Memory usage	🔴 Large	🔴 Large	🟢 Small	🔴 Large	🟢 Small
Performance	🟡 Normal	🟡 Normal	🟢 Fast	🟡 Normal	🟢 Fast
Quality loss	🟢 None	🟡 Slight	🔴 Moderate	🟢 None	🔴 Moderate
Load time	🔴 Slow	🔴 Slow	🟢 Fast	🟡 Normal	🟡 Normal

Estimated memory usage for a single RGBA8 texture with mipmaps enabled:

Texture size	Lossless	Lossy	VRAM Compressed	VRAM Uncompressed	Basis Universal
128×128	🟢 85 KiB	🟢 85 KiB	🟢 21 KiB	🟢 85 KiB	🟢 21 KiB
256×256	🟢 341 KiB	🟢 341 KiB	🟢 85 KiB	🟢 341 KiB	🟢 85 KiB
512×512	🟢 1.33 MiB	🟢 1.33 MiB	🟢 341 KiB	🟢 1.33 MiB	🟢 341 KiB
1024×1024	🟡 5.33 MiB	🟡 5.33 MiB	🟢 1.33 MiB	🟡 5.33 MiB	🟢 1.33 MiB

Texture size	Lossless	Lossy	VRAM Compressed	VRAM Uncompressed	Basis Universal
2048×2048	🔴 21.33 MiB	🔴 21.33 MiB	🟢 5.33 MiB	🔴 21.33 MiB	🟢 5.33 MiB
4096×4096	🔴 85.33 MiB	🔴 85.33 MiB	🔴 21.33 MiB	🔴 85.33 MiB	🔴 21.33 MiB

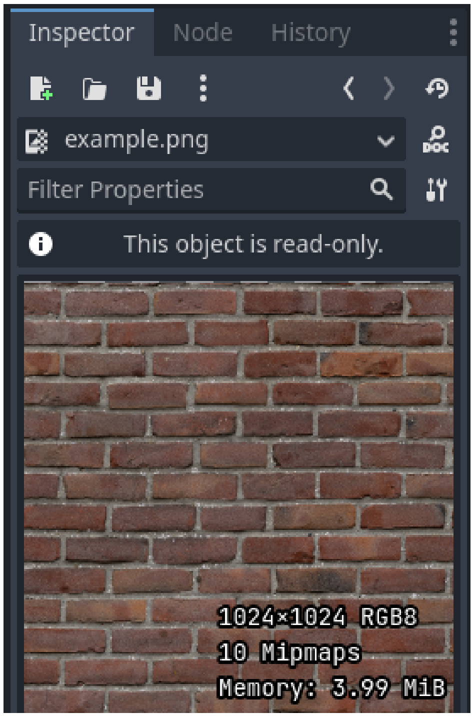
Note

In the above table, memory usage will be reduced by 25% for images that do not have an alpha channel (RGB8). Memory usage will be further decreased by 25% for images that have mipmaps disabled.

Notice how at larger resolutions, the impact of VRAM compression is much greater. With a 4:1 compression ratio (6:1 for opaque textures with S3TC), VRAM compression effectively allows a texture to be twice as large on each axis, while using the same amount of memory on the GPU.

VRAM compression also reduces the memory bandwidth required to sample the texture, which can speed up rendering in memory bandwidth-constrained scenarios (which are frequent on integrated graphics and mobile). These factors combined make VRAM compression a must-have for 3D games with high-resolution textures.

You can preview how much memory a texture takes by double-clicking it in the FileSystem dock, then looking at the Inspector:



example.png

This object is read-only.

1024×1024 RGB8
10 Mipmaps
Memory: 3.99 MiB

Previewing a texture in the Inspector. Credit: Red Brick 03 - Poly Haven [https://polyhaven.com/a/red_brick_03]

Compress > High Quality

Note

High-quality VRAM texture compression is only supported in the Forward+ and Forward Mobile rendering methods.

When using the Compatibility rendering method, this option is always considered disabled.

If enabled, uses BPTC compression on desktop platforms and ASTC compression on mobile platforms. When using BPTC, BC7 is used for SDR textures and BC6H is used for HDR textures.

If disabled (default), uses the faster but lower-quality S3TC compression on desktop platforms and ETC2 on mobile/web platforms. When using S3TC, DXT1 (BC1) is used for opaque textures and DXT5 (BC3) is used for transparent or normal map (RGTC) textures.

BPTC and ASTC support VRAM compression for HDR textures, but S3TC and ETC2 do not (see **HDR Compression** below).

Compress > HDR Compression

Note

This option only has an effect on textures that are imported as HDR formats in Godot (`.hdr` and `.exr` files).

If set to **Disabled**, never uses VRAM compression for HDR textures, regardless of whether they're opaque or transparent. Instead, the texture is converted to RGBE9995 (9-bits per channel + 5-bit exponent = 32 bits per pixel) to reduce memory usage compared to a half-float or single-precision float image format.

If set to **Opaque Only** (default), only uses VRAM compression for opaque HDR textures. This is due to a limitation of HDR formats, as there is no VRAM-compressed HDR format that supports transparency at the same time.

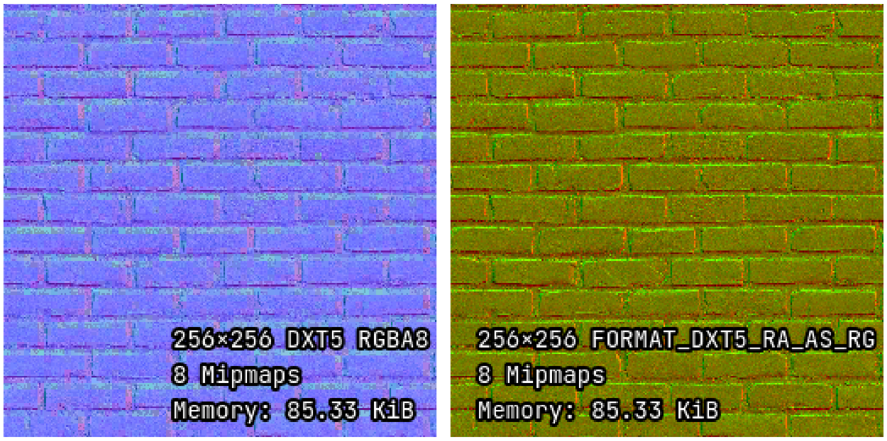
If set to **Always**, will force VRAM compression even for HDR textures with an alpha channel. To perform this, the alpha channel is discarded on import.

Compress > Normal Map

When using a texture as normal map, only the red and green channels are required. Given regular texture compression algorithms produce artifacts that don't look that nice in normal maps, the RGTC compression format is the best fit for this data. Forcing this option to **Enable** will make Godot import the image as RGTC compressed. By default, it's set to **Detect**. This means that if the texture is ever detected to be used as a normal map, it will be changed to **Enable** and reimported automatically.

Note that RGTC compression affects the resulting normal map image. You will have to adjust custom shaders that use the normal map's blue channel to take this into account. Built-in material shaders already ignore the blue channel in a normal map (regardless of the actual normal map's contents).

In the example below, the normal map with RGTC compression is able to preserve its detail much better, while using the same amount of memory as a standard RGBA VRAM-compressed texture:



Normal map with standard VRAM compression (left) and with RGTC VRAM compression (right)

Note

Godot requires the normal map to use the X+, Y+ and Z+ coordinates, which is known as an OpenGL-style normal map. If you've imported a material made to be used with another engine, it may be DirectX-style. In this case, the normal map needs to be converted by enabling the **Normal Map Invert Y** import option.

More information about normal maps (including a coordinate order table for popular engines) can be found here [http://wiki.polycount.com/wiki/Normal_Map_Technical_Details].

Compress > Channel Pack

If set to **sRGB Friendly** (default), prevents the RG color format from being used as it does not support sRGB color.

If set to **Optimized**, allows the RG color format to be used if the texture does not use the blue channel.

A third option **Normal Map (RG Channels)** is *only* available in layered textures (Cubemap, CubemapArray, Texture2DArray and Texture3D). This forces all layers from the texture to be imported with the RG color format to reduce memory usage, with only the red and green channels preserved. This only has an effect on textures with the **VRAM Compressed** or **Basis Universal** compression modes.

Mipmaps > Generate

If enabled, smaller versions of the texture are generated on import. For example, a 64×64 texture will generate 6 mipmaps (32×32, 16×16, 8×8, 4×4, 2×2, 1×1). This has several benefits:

- Textures will not become grainy in the distance (in 3D), or if scaled down due to camera zoom or CanvasItem scale (in 2D).
- Performance will improve if the texture is displayed in the distance, since sampling smaller versions of the original texture is faster and requires less memory bandwidth.

The downside of mipmaps is that they increase memory usage by roughly 33%.

It's recommended to enable mipmaps in 3D. However, in 2D, this should only be enabled if your project visibly benefits from having mipmaps enabled. If the camera never zooms out significantly, there won't be a benefit to enabling mipmaps but memory usage will increase.

Mipmaps > Limit

Warning

Mipmaps > Limit is currently not implemented and has no effect when changed.

If set to a value greater than `-1`, limits the maximum number of mipmaps that can be generated. This can be decreased if you don't want textures to become too low-resolution at extreme distances, at the cost of some graininess.

Roughness > Mode

The color channel to consider as a roughness map in this texture. Only effective if **Roughness > Src Normal** is not empty.

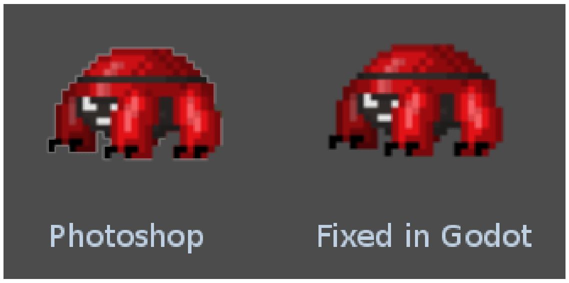
Roughness > Src Normal

The path to the texture to consider as a normal map for roughness filtering on import. Specifying this can help decrease specular aliasing slightly in 3D.

Roughness filtering on import is only used in 3D rendering, not 2D.

Process > Fix Alpha Border

This puts pixels of the same surrounding color in transition from transparent to opaque areas. For textures displayed with bilinear filtering, this helps mitigate the outline effect when exporting images from an image editor.



It's recommended to leave this enabled (as it is by default), unless this causes issues for a particular image.

Process > Premult Alpha

An alternative to fixing darkened borders with **Fix Alpha Border** is to use premultiplied alpha. By enabling this option, the texture will be converted to this format. A premultiplied alpha texture requires specific materials to be displayed correctly:

- In 2D, a CanvasItemMaterial will need to be created and configured to use the **Premul Alpha** blend mode on CanvasItems that use this texture.
- In 3D, there is no support for premultiplied alpha blend mode yet, so this option is only suited for 2D.

Process > Normal Map Invert Y

Godot requires the normal map to use the X+, Y+ and Z+ coordinates, which is known as an OpenGL-style normal map. If you've imported a material made to be used with another engine, it may be DirectX-style. In this case, the normal map needs to be converted by enabling the **Normal Map Invert Y** import option.

More information about normal maps (including a coordinate order table for popular engines) can be found here [http://wiki.polycount.com/wiki/Normal_Map_Technical_Details].

Process > HDR as sRGB

Some HDR images you can find online may be broken and contain sRGB color data (instead of linear color data). It is advised not to use those files. If you absolutely have to, enabling this option on will make them look correct.

Warning

Enabling **HDR as sRGB** on well-formatted HDR images will cause the resulting image to look too dark, so leave this disabled if unsure.

Process > HDR Clamp Exposure

Some HDR panorama images you can find online may contain extremely bright pixels, due to being taken from real life sources without any clipping.

While these HDR panorama images are accurate to real life, this can cause the radiance map generated by Godot to contain sparkles when used as a background sky. This can be seen in material reflections (even on rough materials in extreme cases). Enabling **HDR Clamp Exposure** can resolve this using a smart clamping formula that does not introduce *visible* clipping – glow will keep working when looking at the background sky.

Process > Size Limit

If set to a value greater than `0`, the size of the texture is limited on import to a value smaller than or equal to the value specified here. For non-square textures, the size limit affects the longer dimension, with the shorter dimension scaled to preserve aspect ratio. Resizing is performed using cubic interpolation.

This can be used to reduce memory usage without affecting the source images, or avoid issues with textures not displaying on mobile/web platforms (as these usually can't display textures larger than 4096×4096).

Detect 3D > Compress To

This changes the Compress > Mode option that is used when a texture is detected as being used in 3D.

Changing this import option only has an effect if a texture is detected as being used in 3D. Changing this to **Disabled** then reimporting will not change the existing compress mode on a texture (if it's detected to be used in 3D), but choosing **VRAM Compressed** or **Basis Universal** will.

SVG > Scale

This is only available for SVG images.

The scale the SVG should be rendered at, with `1.0` being the original design size. Higher values result in a larger image. Note that unlike font oversampling, this affects the physical size the SVG is rendered at in 2D. See also **Editor > Scale With Editor Scale** below.

Editor > Scale With Editor Scale

This is only available for SVG images.

If true, scales the imported image to match the editor's display scale factor. This should be enabled for editor plugin icons and custom class icons, but should be left disabled otherwise.

Editor > Convert Colors With Editor Theme

This is only available for SVG images.

If checked, converts the imported image's colors to match the editor's icon and font color palette. This assumes the image uses the exact same colors as Godot's own color palette for editor icons, with the source file designed for a dark editor theme. This should be enabled for editor plugin icons and custom class icons, but should be left disabled otherwise.

Importing SVG images with text

As the SVG library used in Godot doesn't support rasterizing text found in SVG images, text must be converted to a path first. Otherwise, text won't appear in the rasterized image.

There are two ways to achieve this in a non-destructive manner, so you can keep editing the original text afterwards:

- Select your text object in Inkscape, then duplicate it in place by pressing `Ctrl + D` and use **Path > Object to Path**. Hide the original text object afterwards using the **Layers and Objects** dock.
- Use the Inkscape command line to export an SVG from another SVG file with text converted to paths:

```
inkscape --export-text-to-path --export-filename
svg_with_text_converted_to_path.svg svg_with_text.svg
```

Best practices

Supporting high-resolution texture sizes in 2D without artifacts

To support multiple resolutions with crisp visuals at high resolutions, you will need to use high-resolution source images (suited for the highest resolution you wish to support without blurriness, which is typically 4K in modern desktop games).

There are 2 ways to proceed:

- Use a high base resolution in the project settings (such as 4K), then use the textures at original scale. This is an easier approach.
- Use a low base resolution in the project settings (such as 1080p), then downscale textures when using them. This is often more difficult and can make various calculations in script tedious, so the approach described above is recommended instead.

After doing this, you may notice that textures become grainy at lower viewport resolutions. To resolve this, enable **Mipmaps** on textures used in 2D in the Import dock. This will increase memory usage.

Enabling mipmaps can also make textures appear blurrier, but you can choose to make textures sharper (at the cost of some graininess) by setting **Rendering > Textures > Default Filters > Texture Mipmap Bias** to a negative value.

Use appropriate texture sizes in 3D

While there's no "one size fits all" recommendation, here are some general recommendations for choosing texture sizes in 3D:

- The size of a texture should be adjusted to have a consistent texel density compared to surrounding objects. While this cannot be ensured perfectly when sticking to power-of-two texture sizes, it's usually possible to keep texture detail fairly consistent throughout a 3D scene.
- The smaller the object appears on screen, the smaller its texture should be. For example, a tree that only appears in the background doesn't need a texture resolution as high as other objects the player may be able to walk close to.
- Using power-of-two texture sizes is recommended, but is not required. Textures don't have to be square – sizes such as 1024×512 are acceptable.
- There are diminishing returns to using large texture sizes, despite the increased memory usage and loading times. Most modern 3D games not using a pixel art style stick to 2048×2048 textures on average, with 1024×1024 and 512×512 for textures spanning smaller surfaces.
- When working with physically-based materials in 3D, you can reduce memory usage and file size without affecting quality too much by using a lower resolution for certain texture maps. This works especially well for textures that only feature low-frequency detail (such as a normal map for a snow texture).

If you have control over how the 3D models are created, these tips are also worth exploring:

- When working with 3D models that are mostly symmetrical, you may be able to use mirrored UVs to double the effective texel density. This may look unnatural when used on human faces though.
- When working with 3D models using a low-poly style and plain colors, you can rely on vertex colors instead of textures to represent colors on the model's surfaces.

See also

Images can be loaded and saved at runtime using runtime file loading and saving, including from an exported project.

Importing audio samples

Supported audio formats

Godot provides 3 options to import your audio data: WAV, Ogg Vorbis and MP3.

Each format has different advantages:

- WAV files use raw data or light compression (IMA-ADPCM or QOA). They are lightweight to play back on the CPU (hundreds of simultaneous voices in this format are fine). The downside is that they take up a lot of disk space.
- Ogg Vorbis files use a stronger compression that results in much smaller file size, but require significantly more processing power to play back.
- MP3 files use better compression than WAV with IMA-ADPCM or QOA, but worse than Ogg Vorbis. This means that an MP3 file with roughly equal quality to Ogg Vorbis will be significantly larger. On the bright side, MP3 requires less CPU usage to play back compared to Ogg Vorbis.

Note

If you've compiled the Godot editor from source with specific modules disabled, some formats may not be available.

Here is a comparative chart representing the file size of 1 second of audio with each format:

Format	1 second of audio
WAV 24-bit, 96 kHz, stereo	576 KB
WAV 16-bit, 44 kHz, mono	88 KB
WAV IMA-ADPCM, 44 kHz, mono	22 KB
WAV QOA, 44 kHz, mono	17 KB
MP3 192 Kb/s, stereo	24 KB
Ogg Vorbis 128 Kb/s, stereo	16 KB
Ogg Vorbis 96 Kb/s, stereo	12 KB

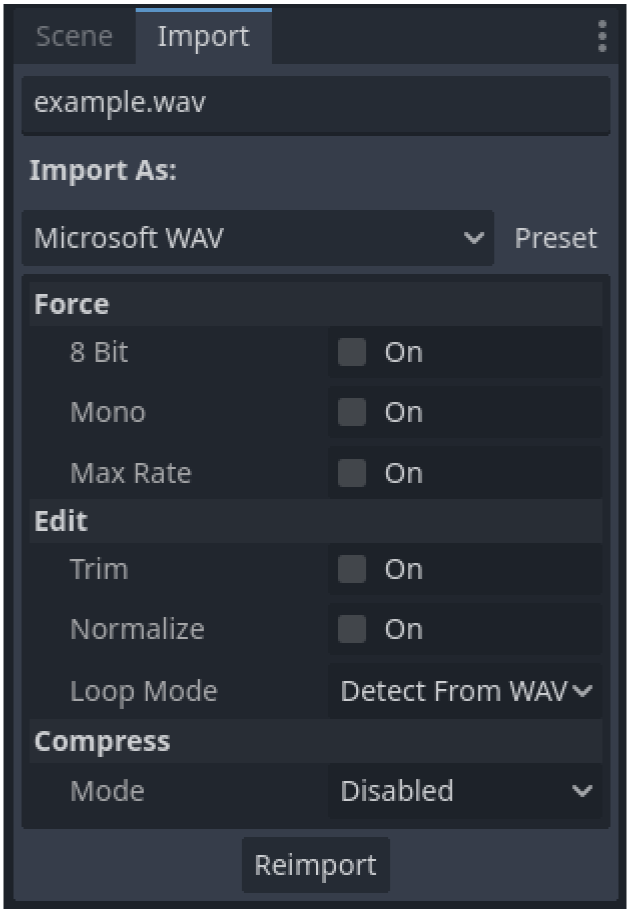
Note that the MP3 and Ogg Vorbis figures can vary depending on the encoding type. The above figures use CBR encoding for simplicity, but most Ogg Vorbis and MP3 files you can find online are encoded with VBR encoding which is more efficient. VBR encoding makes the effective audio file size depend on how "complex" the source audio is.

Tip

Consider using WAV for short and repetitive sound effects, and Ogg Vorbis for music, speech, and long sound effects. MP3 is useful for mobile and web projects where CPU resources are limited, especially when playing multiple compressed sounds at the same time (such as long ambient sounds).

Importing audio samples

Several options are available in the Import dock after selecting a WAV file in the FileSystem dock:



example.wav

Import As: Microsoft WAV

Force

- 8 Bit ☐ On
- Mono ☐ On
- Max Rate ☐ On

Edit

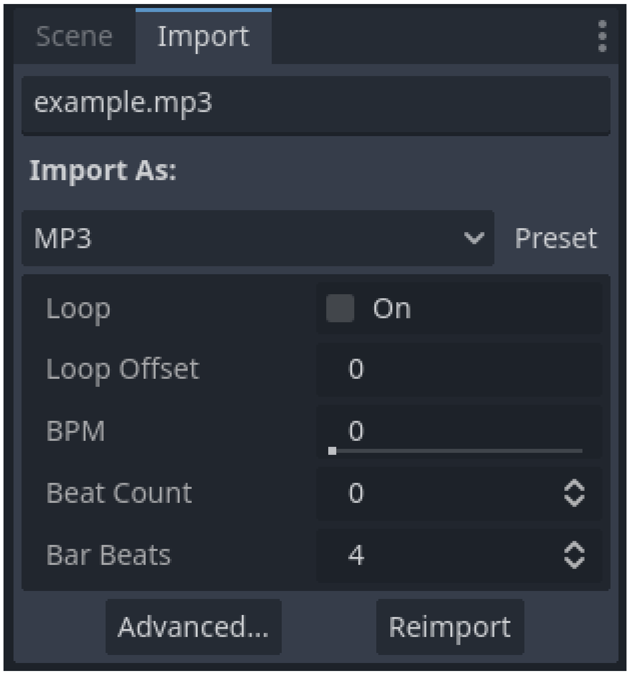
- Trim ☐ On
- Normalize ☐ On
- Loop Mode: Detect From WAV

Compress

- Mode: Disabled

Import options in the Import dock after selecting a WAV file in the FileSystem dock

The set of options available after selecting an Ogg Vorbis or MP3 file is different:



example.mp3

MP3

Loop ☐ On

Loop Offset 0

BPM 0

Beat Count 0

Bar Beats 4

Import options in the Import dock after selecting an MP3 file in the FileSystem dock. Options are identical for Ogg Vorbis files.

After importing a sound, you can play it back using the AudioStreamPlayer, AudioStreamPlayer2D or AudioStreamPlayer3D nodes. See Audio streams for more information.

Import options (WAV)

Force > 8 Bit

If enabled, forces the imported audio to use 8-bit quantization if the source file is 16-bit or higher.

Enabling this is generally not recommended, as 8-bit quantization decreases audio quality significantly. If you need smaller file sizes, consider using Ogg Vorbis or MP3 audio instead.

Force > Mono

If enabled, forces the imported audio to be mono if the source file is stereo. This decreases the file size by 50% by merging the two channels into one.

Force > Max Rate

If set to a value greater than `0`, forces the audio's sample rate to be reduced to a value lower than or equal to the value specified here.

This can decrease file size noticeably on certain sounds, without impacting quality depending on the actual sound's contents. See Best practices for more information.

Edit > Trim

The source audio file may contain long silences at the beginning and/or the end. These silences are inserted by DAWs when saving to a waveform, which increases their size unnecessarily and add latency to the moment they are played back.

Enabling **Trim** will automatically trim the beginning and end of the audio if it's lower than -50 dB *after* normalization (see **Edit > Normalize** below). A fade-in/fade-out period of 500 samples is also used during trimming to avoid audible pops.

Edit > Normalize

If enabled, audio volume will be *normalized* so that its peak volume is equal to 0 dB. When enabled, normalization will make audio sound louder depending on its original peak volume.

Edit > Loop Mode

Unlike Ogg Vorbis and MP3, WAV files can contain metadata to indicate whether they're looping (in addition to loop points). By default, Godot will follow this metadata, but you can choose to apply a specific loop mode:

- **Disabled:** Don't loop audio, even if metadata indicates the file should be played back looping.
- **Forward:** Standard audio looping.
- **Ping-Pong:** Play audio forward until it's done playing, then play it backward and repeat. This is similar to mirrored texture repeat, but for audio.
- **Backward:** Play audio in reverse and loop back to the end when done playing.

When choosing one of the **Forward**, **Ping-Pong** or **Backward** loop modes, loop points can also be defined to make only a specific part of the sound loop. **Loop Begin** is set in samples after the beginning of the audio file. **Loop End** is also set in samples after the beginning of the audio file, but will use the end of the audio file if set to `-1`.

Warning

In AudioStreamPlayer, the `finished` signal won't be emitted for looping audio when it reaches the end of the audio file, as the audio will keep playing indefinitely.

Compress > Mode

Three compression modes can be chosen from for WAV files: **Disabled** (default), **RAM (Ima-ADPCM)**, or **QOA (Quite OK**

Audio). **RAM (Ima-ADPCM)** reduces file size and memory usage a little, at the cost of decreasing quality in an audible manner. **QOA (Quite OK Audio)** reduces file size a bit more than **RAM (Ima-ADPCM)** and the quality decrease is much less noticeable, at the cost of higher CPU usage (still much lower than MP3).

Ogg Vorbis and MP3 don't decrease quality as much and can provide greater file size reductions, at the cost of higher CPU usage during playback. This higher CPU usage is usually not a problem (especially with MP3), unless playing dozens of compressed sounds at the same time on mobile/web platforms.

Import options (Ogg Vorbis and MP3)

Loop

If enabled, the audio will begin playing at the beginning after playback ends by reaching the end of the audio.

Warning

In AudioStreamPlayer, the `finished` signal won't be emitted for looping audio when it reaches the end of the audio file, as the audio will keep playing indefinitely.

Loop Offset

The loop offset determines where audio will start to loop after playback reaches the end of the audio. This can be used to only loop a part of the audio file, which is useful for some ambient sounds or music. The value is determined in seconds relative to the beginning of the audio, so `0` will loop the entire audio file.

Only has an effect if **Loop** is enabled.

A more convenient editor for **Loop Offset** is provided in the Advanced import settings dialog, as it lets you preview your changes without having to reimport the audio.

BPM

The Beats Per Minute of the audio track. This should match the BPM measure that was used to compose the track. This is only relevant for music that wishes to make use of interactive music functionality, not sound effects.

A more convenient editor for **BPM** is provided in the Advanced import settings dialog, as it lets you preview your changes without having to reimport the audio.

Beat Count

The beat count of the audio track. This is only relevant for music that wishes to make use of interactive music functionality, not sound effects.

A more convenient editor for **Beat Count** is provided in the Advanced import settings dialog, as it lets you preview your changes without having to reimport the audio.

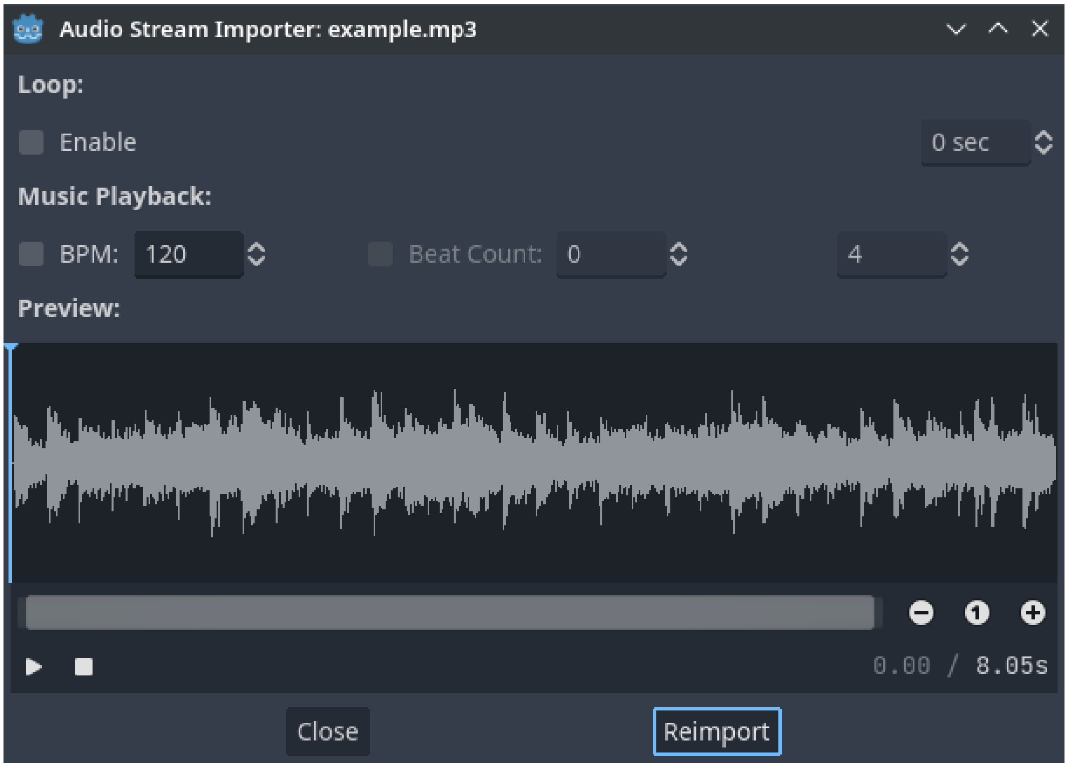
Bar Beats

The number of bars within a single beat in the audio track. This is only relevant for music that wishes to make use of interactive music functionality , not sound effects.

A more convenient editor for **Bar Beats** is provided in the Advanced import settings dialog, as it lets you preview your changes without having to reimport the audio.

Advanced import settings (Ogg Vorbis and MP3)

If you double-click an Ogg Vorbis or MP3 file in the FileSystem dock (or choose **Advanced...** in the Import dock), you will see a dialog appear:


Audio Stream Importer: example.mp3

Loop:

☐ Enable 0 sec

Music Playback:

☐ BPM: 120 ☐ Beat Count: 0 4

Preview:

0.00 / 8.05s

Advanced dialog when double-clicking an Ogg Vorbis or MP3 file in the FileSystem dock

This dialog allows you to edit the audio's loop point with a real-time preview, in addition to the BPM, beat count and bar beats. These 3 settings are currently unused, but they will be used in the future for interactive music support (which allows smoothly transitioning between different music tracks).

Note

Unlike WAV files, Ogg Vorbis and MP3 only support a "loop begin" loop point, not a "loop end" point. Looping can also be only be standard forward looping, not ping-pong or backward.

Best practices

Use appropriate quality settings

While keeping pristine-quality audio sources is important if you're performing editing, using the same quality in the exported project is not necessary. For WAV files, Godot offers several import options to reduce the final file size without modifying the source file on disk.

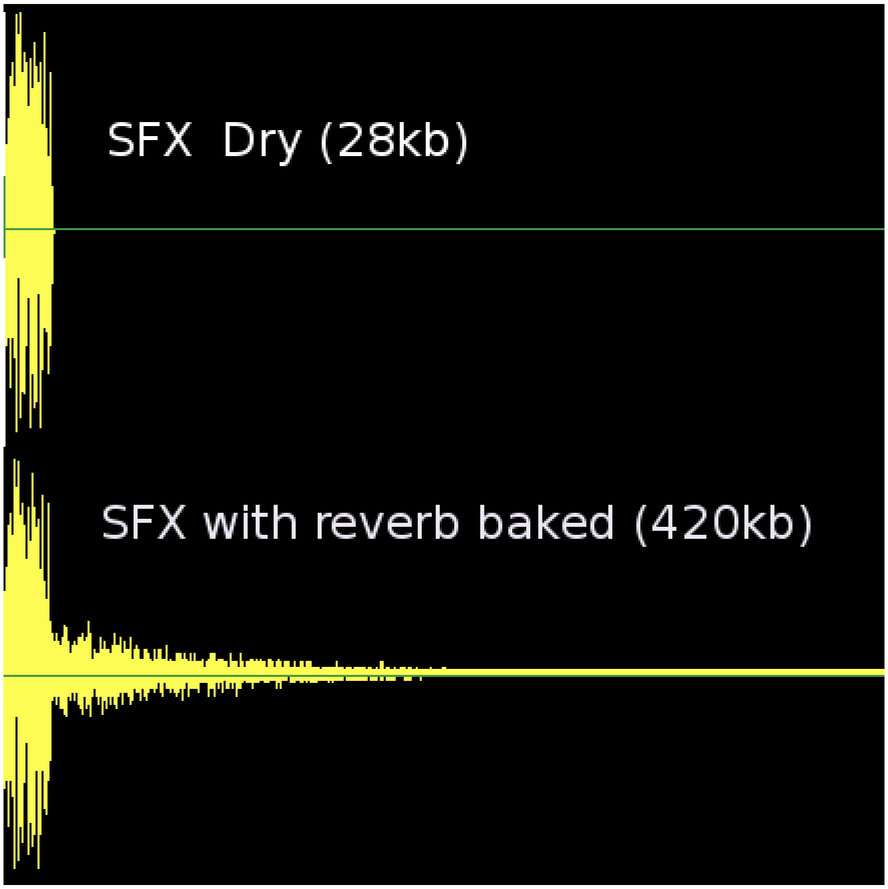
To reduce memory usage and file size, choose an appropriate quantization, sample rate and number of channels for your audio:

- There's no *audible* benefit to using 24-bit audio, especially in a game where several sounds are often playing at the same time (which makes it harder to appreciate individual sounds).
- Unless you are slowing down the audio at run-time, there's no *audible* benefit to using a sample rate greater than 48 kHz. If you wish to keep a source with a higher sample rate for editing, use the **Force > Max Rate** import option to limit the sample rate of the imported sound (only available for WAV files).
- Many sound effects can generally be converted to mono as opposed to stereo. If you wish to keep a source with stereo for editing, use the **Force > Mono** import option to convert the imported sound to mono (only available for WAV files).

- Voices can generally be converted to mono, but can also have their sample rate reduced to 22 kHz without a noticeable loss in quality (unless the voice is very high-pitched). This is because most human voices never go past 11 kHz.

Use real-time audio effects to reduce file size

Godot has an extensive bus system with built-in effects. This saves SFX artists the need to add reverb to the sound effects, reducing their size greatly and ensuring correct trimming.



As you can see above, sound effects become much larger in file size with reverb added.

See also

Audio samples can be loaded and saved at runtime using runtime file loading and saving, including from an exported project.

Importing translations

Games and internationalization

The gaming community isn't monolingual or monocultural. It's made up of many different languages and cultures - just like the Godot community! If you want to allow players to experience your game in their language, one of things you'll need to provide is text translations, which Godot supports via internationalized text.

In regular desktop or mobile applications, internationalized text is usually located in resource files (or .po files for GNU stuff). Games, however, can use several orders of magnitude more text than applications, so they must support efficient methods for dealing with loads of multilingual text.

There are two approaches to generate multilingual language games and applications. Both are based on a key:value system. The first is to use one of the languages as the key (usually English), the second is to use a specific identifier. The first approach is probably easier for development if a game is released first in English, later in other languages, but a complete nightmare if working with many languages at the same time.

In general, games use the second approach and a unique ID is used for each string. This allows you to revise the text while it is being translated to other languages. The unique ID can be a number, a string, or a string with a number (it's just a unique string anyway).

Note

If you need a more powerful file format, Godot also supports loading translations written in the gettext `.po` format. See

Localization using gettext for details.

Translation format

To complete the picture and allow efficient support for translations, Godot has a special importer that can read CSV files. Most spreadsheet editors can export to this format, so the only requirement is that the files have a special arrangement. The CSV files **must** be saved with UTF-8 encoding without a byte order mark [https://en.wikipedia.org/wiki/Byte_order_mark].

CSV files must be formatted as follows:

keys	<lang1>	<lang2>	<langN>
KEY1	string	string	string
KEY2	string	string	string
KEYN	string	string	string

The "lang" tags must represent a language, which must be one of the valid locales supported by the engine, or they must start with an underscore (_), which means the related column is served as comment and won't be imported. The "KEY" tags must be unique and represent a string universally (they are usually in uppercase, to differentiate from other strings). These keys will be replaced at runtime by the matching translated string. Note that the case is important, "KEY1" and "Key1" will be different keys. The top-left cell is ignored and can be left empty or having any content. Here's an example:

keys	**en**	**es**	**ja**
GREET	Hello, friend!	Hola, amigo!	こんにちは
ASK	How are you?	Cómo está?	元気ですか
BYE	Goodbye	Adiós	さようなら
QUOTE	"Hello" said the man.	"Hola" dijo el hombre.	「こんにちは」男は言いました

The same example is shown below as a comma-separated plain text file, which should be the result of editing the above in a spreadsheet. When editing the plain text version, be sure to enclose with double quotes any message that contains commas, line breaks or double quotes, so that commas are not parsed as delimiters, line breaks don't create new entries and double quotes are not parsed as enclosing characters. Be sure to escape any double quotes a message may contain by preceding them with another double quote. Alternatively, you can select another delimiter than comma in the import options.

```
keys,en,es,ja
GREET,"Hello, friend!","Hola, amigo!",こんにちは
ASK,How are you?,Cómo está?,元気ですか
BYE,Goodbye,Adiós,さようなら
```

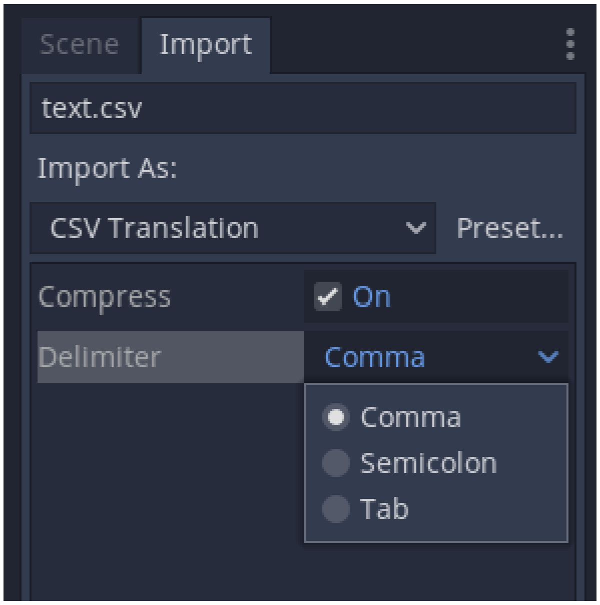
```
QUOTE,"""Hello"" said the man.","""Hola"" dijo el hombre.",「こんにちは」男は言いました
```

CSV importer

Godot will treat CSV files as translations by default. It will import them and generate one or more compressed translation resource files next to it.

Importing will also add the translation to the list of translations to load when the game runs, specified in project.godot (or the project settings). Godot allows loading and removing translations at runtime as well.

Select the `.csv` file and access the **Import** dock to define import options. You can toggle the compression of the imported translations, and select the delimiter to use when parsing the CSV file.



text.csv

Import As: CSV Translation

Compress ☑ On

Delimiter Comma

Be sure to click **Reimport** after any change to these options.

Importing 3D scenes

Godot supports importing 3D scenes from various file formats. This documentation section describes what those formats are, and how to use them, including exporting with the correct conventions and best practices, and how to customize the node type using a suffix in the node name. The import configuration article describes how to customize the imported data using the import dock, the advanced import settings dialog, and inherited scenes.

- Available 3D formats
- Model export considerations
- Node type customization using name suffixes
- Import configuration

See also

3D scenes can be loaded at runtime using runtime file loading and saving, including from an exported project.

Available 3D formats

When dealing with 3D assets, Godot has a flexible and configurable importer.

Godot works with *scenes*. This means that the entire scene being worked on in your favorite 3D modeling software will be transferred as close as possible.

Godot supports the following 3D *scene file formats*:

- glTF 2.0 **(recommended)**. Godot has support for both text (`.gltf`) and binary (`.glb`) formats.
- `.blend` (Blender). This works by calling Blender to export to glTF in a transparent manner (requires Blender to be installed).
- DAE (COLLADA), an older format that is supported.
- OBJ (Wavefront) format + their MTL material files. This is also supported, but pretty limited given the format's limitations (no support for pivots, skeletons, animations, UV2, PBR materials, ...).
- FBX, supported via the ufbx [https://github.com/ufbx/ufbx] library. The previous import workflow used FBX2glTF [https://github.com/godotengine/FBX2glTF] integration. This requires installing an external program that links against the proprietary FBX SDK, so we recommend using the default ubfx method or other formats listed above (if suitable for your workflow).

Copy the scene file together with the textures and mesh data (if separate) to the project repository, then Godot will do a full import when focusing the editor window.

Exporting glTF 2.0 files from Blender (recommended)

There are 3 ways to export glTF files from Blender:

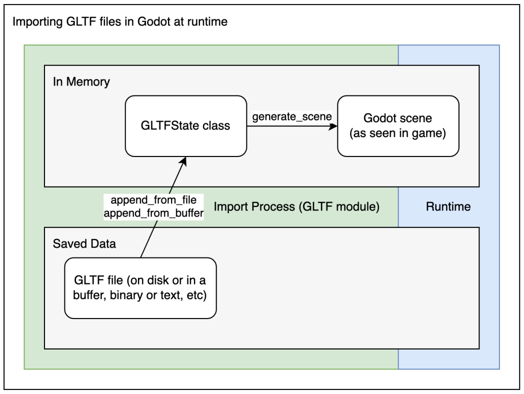
- As a glTF binary file (`.glb`).
- As a glTF text-based file with embedded binary data (`.gltf` file)
- As a glTF text-based file with separate binary data and textures (`.gltf` file + `.bin` file + textures).

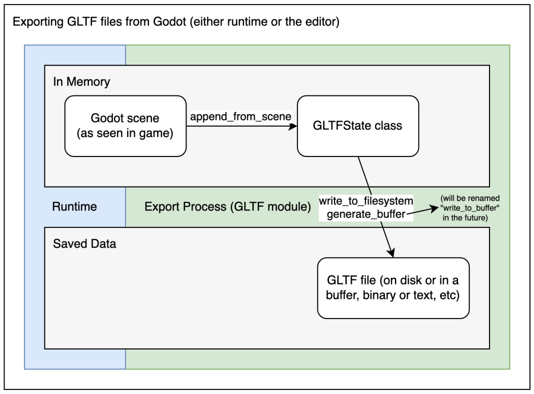
glTF binary files (`.glb`) are the smallest of the three options. They include the mesh and textures set up in Blender. When brought into Godot the textures are part of the object's material file.

glTF embedded files (`.gltf`) function the same way as binary files. They don't provide extra functionality in Godot, and shouldn't be used since they have a larger file size.

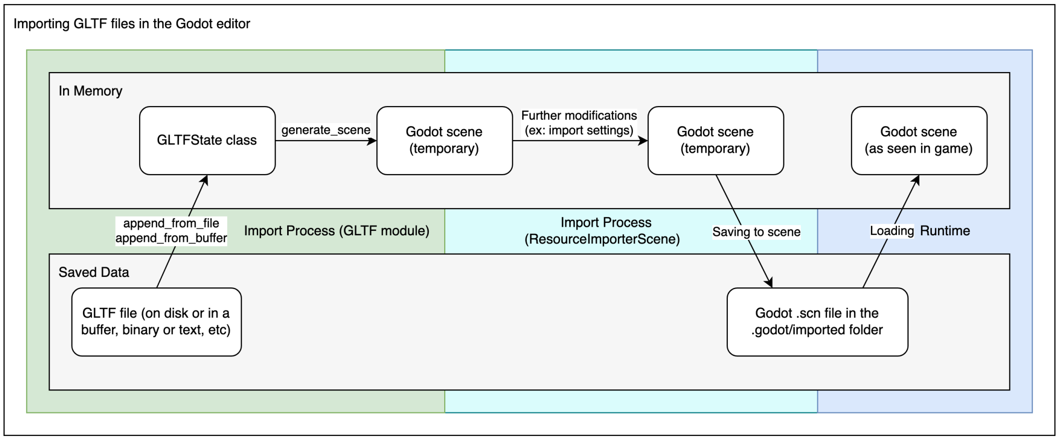
There are two reasons to use glTF with the textures separate. One is to have the scene description in a text based format and the binary data in a separate binary file. This can be useful for version control if you want to review changes in a text-based format. The second is you need the texture files separate from the material file. If you don't need either of those, glTF binary files are fine.

The glTF import process first loads the glTF file's data into an in-memory GLTFState class. This data is then used to generate a Godot scene. When importing files at runtime, this scene can be directly added to the tree. The export process is the reverse of this, a Godot scene is converted to a GLTFState class, then the glTF file is generated from that.


Importing GLTF files in Godot at runtime
In Memory
GLTFState class
generate_scene
Godot scene (as seen in game)
append_from_file
append_from_buffer
Import Process (GLTF module)
Runtime
Saved Data
GLTF file (on disk or in a buffer, binary or text, etc)


Exporting GLTF files from Godot (either runtime or the editor)
In Memory
Godot scene (as seen in game)
append_from_scene
GLTFState class
Runtime
Export Process (GLTF module)
write_to_filesystem
generate_buffer
(will be renamed "write_to_buffer" in the future)
Saved Data
GLTF file (on disk or in a buffer, binary or text, etc)

When importing glTF files in the editor, there are two more steps. After generating the Godot scene, the ResourceImporterScene class is used to apply additional import settings, including settings you set through the Import dock and the Advanced Import Settings dialog. This is then saved as a Godot scene file, which is what gets used when you run/export your game.


Importing GLTF files in the Godot editor
In Memory
GLTFState class
generate_scene
Godot scene (temporary)
Further modifications (ex: import settings)
Godot scene (temporary)
Godot scene (as seen in game)
append_from_file
append_from_buffer
Import Process (GLTF module)
Import Process (ResourceImporterScene)
Saving to scene
Loading
Runtime
Saved Data
GLTF file (on disk or in a buffer, binary or text, etc)
Godot .scn file in the .godot/imported folder

Warning

If your model contains blend shapes (also known as "shape keys" and "morph targets"), your glTF export setting **Export Deformation Bones Only** needs to be configured to **Enabled** under the Animation export configurations.

Exporting non-deforming bones anyway will lead to incorrect shading.

Note

Blender versions older than 3.2 do not export emissive textures with the glTF file. If your model uses one and you're using an older version of Blender, it must be brought in separately.

By default, Blender has backface culling disabled on materials and will export materials to match how they render in Blender. This means that materials in Godot will have their cull mode set to **Disabled**. This can decrease performance since backfaces will be rendered, even when they are being culled by other faces. To resolve this, enable **Backface Culling** in Blender's Materials tab, then export the scene to glTF again.

Importing `.blend` files directly within Godot

Note

This functionality requires Blender 3.0 or later. For best results, we recommend using Blender 3.5 or later, as it includes many fixes to the glTF exporter.

It is **strongly** recommended to use an official Blender release downloaded from blender.org, as opposed to a Linux distribution package or Flatpak. This avoids any issues related to packaging, such as different library versions that can cause incompatibilities or sandboxing restrictions.

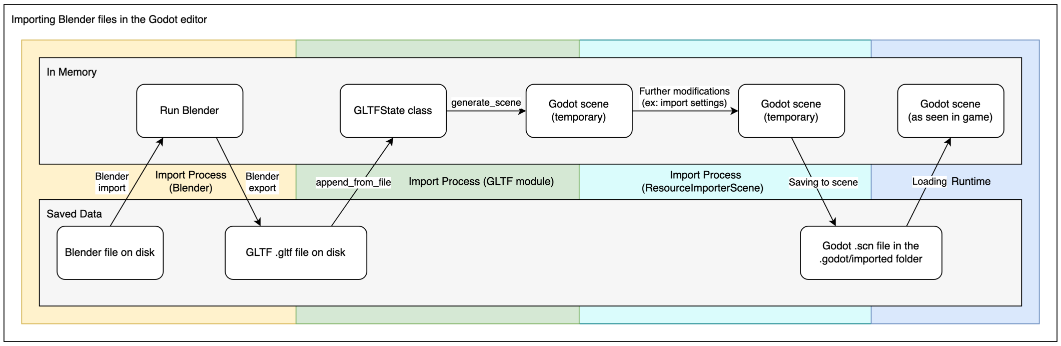
From Godot 4.0 onwards, the editor can directly import `.blend` files by calling [Blender](https://www.blender.org/) [https://www.blender.org/]'s glTF export functionality in a transparent manner.

This allows you to iterate on your 3D scenes faster, as you can save the scene in Blender, alt-tab back to Godot then see your changes immediately. When working with version control, this is also more efficient as you no longer need to commit a copy of the exported glTF file to version control.

To use `.blend` import, you must install Blender before opening the Godot editor (if opening a project that already contains `.blend` files). If you keep Blender installed at its default location, Godot should be able to detect its path automatically. If this isn't the case, configure the path to the directory containing the Blender executable in the Editor Settings (**Filesystem > Import > Blender > Blender 3 Path**).

If you keep `.blend` files within your project folder but don't want them to be imported by Godot, disable **Filesystem > Import > Blender > Enabled** in the advanced Project Settings.

The `.blend` import process converts to glTF first, so it still uses Godot's glTF import code. Therefore, the `.blend` import process is the same as the glTF import process, but with an extra step at the beginning.


Importing Blender files in the Godot editor
In Memory
Run Blender
GLTFState class
generate_scene
Godot scene (temporary)
Further modifications (ex: import settings)
Godot scene (temporary)
Godot scene (as seen in game)
Blender import
Import Process (Blender)
Blender export
append_from_file
Import Process (GLTF module)
Import Process (ResourceImporterScene)
Saving to scene
Loading
Runtime
Saved Data
Blender file on disk
GLTF .gltf file on disk
Godot .scn file in the .godot/imported folder

Note

When working in a team, keep in mind using `.blend` files in your project will require *all* team members to have Blender installed. While Blender is a free download, this may add friction when working on the project. `.blend` import is also not available on the Android and web editors, as these platforms can't call external programs.

If this is problematic, consider using glTF scenes exported from Blender instead.

Exporting DAE files from Blender

Blender has built-in COLLADA support, but it does not work properly for the needs of game engines and shouldn't be used as-is. However, scenes exported with the built-in Collada support may still work for simple scenes without animation.

For complex scenes or scenes that contain animations, Godot provides a Blender plugin [https://github.com/godotengine/collada-exporter] that will correctly export COLLADA scenes for use in Godot.

Importing OBJ files in Godot

OBJ is one of the simplest 3D formats out there, so Godot should be able to import most OBJ files successfully. However, OBJ is also a very limited format: it doesn't support skinning, animation, UV2 or PBR materials.

There are 2 ways to use OBJ meshes in Godot:

- Load them directly in a MeshInstance3D node, or any other property that expects as mesh (such as GPUParticles3D). This is the default mode.

- Change their import mode to **OBJ as Scene** in the Import dock then restart the editor. This allows you to use the same import options as glTF or Collada scenes, such as unwrapping UV2 on import (for Using Lightmap global illumination).

Note

Blender 3.4 and later can export RGB vertex colors in OBJ files (this is a nonstandard extension of the OBJ format). Godot is able to import those vertex colors since Godot 4.0, but they will not be displayed on the material unless you enable **Vertex Color > Use As Albedo** on the material.

Vertex colors from OBJ meshes keep their original color space once imported (sRGB/linear), but their brightness is clamped to 1.0 (they can't be overbright).

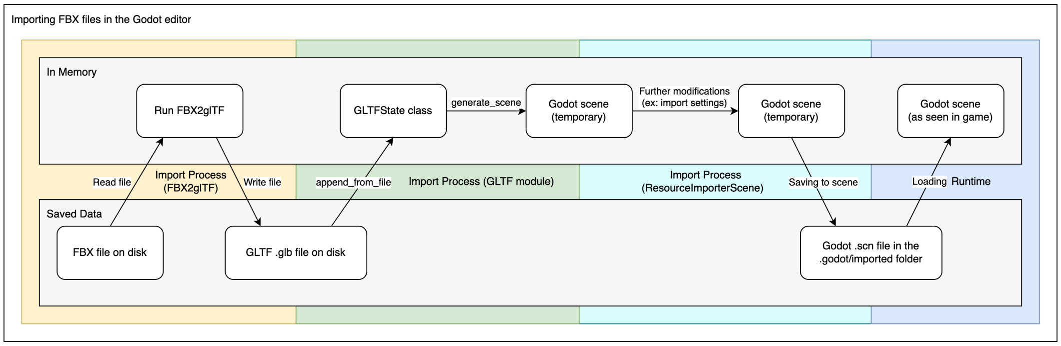
Importing FBX files in Godot

By default any FBX file added to a Godot project in Godot 4.3 or later will use the ufbx import method. Any file that was was added to a project in a previous version, such as 4.2, will continue to be imported via the FBX2glTF method unless you go into that files import settings, and change the importer to `ufbx`.

If you keep `.fbx` files within your project folder but don't want them to be imported by Godot, disable **Filesystem > Import > FBX > Enabled** in the advanced Project Settings.

If you want to setup the FBX2glTF workflow, which is generally not recommend unless you have a specific reason to use it, you need to download the FBX2glTF [https://github.com/godotengine/FBX2glTF] executable, then specify the path to that executable in the editor settings under **Filesystem > Import > FBX > FBX2glTFPath**

The FBX2glTF import process converts to glTF first, so it still uses Godot's glTF import code. Therefore, the FBX import process is the same as the glTF import process, but with an extra step at the beginning.


Importing FBX files in the Godot editor
In Memory
Run FBX2glTF
GLTFState class
generate_scene
Godot scene (temporary)
Further modifications (ex: import settings)
Godot scene (temporary)
Godot scene (as seen in game)
Read file
Import Process (FBX2glTF)
Write file
append_from_file
Import Process (GLTF module)
Import Process (ResourceImporterScene)
Saving to scene
Loading
Runtime
Saved Data
FBX file on disk
GLTF .glb file on disk
Godot .scn file in the .godot/imported folder

See also

The full installation process for using FBX2glTF in Godot is described on the FBX import page of the Godot website [https://godotengine.org/fbx-import].

Model export considerations

Before exporting a 3D model from a 3D modeling application, such as Blender, there are some considerations that should be taken into account to ensure that the model follows the conventions and best practices for Godot.

3D asset direction conventions

Godot uses a right-handed, Y-is-up coordinate system, with the -Z axis as the camera's forward direction. This is the same as OpenGL. This implies that +Z is back, +X is right, and -X is left for a camera.

The convention for 3D assets is to face the opposite direction as the camera, so that characters and other assets are facing the camera by default. This convention is extremely common in 3D modeling applications, and is codified in glTF as part of the glTF 2.0 specification [https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html#coordinate-system-and-units]. This means that for oriented 3D assets (such as characters), the +Z axis is the direction of the front, so -Z is the rear, +X is the left side, and -X is the right side for a 3D asset. In Blender, this means that +Y is rear and -Y is front for an asset.

When rotating an oriented 3D asset in Godot, use the `use_model_front` option on the `look_at` functions, and use the `Vector3.MODEL_*` constants to perform calculations in the oriented asset's local space.

For assets without an intrinsic front side or forward direction, such as a game map or terrain, take note of the cardinal directions instead. The convention in Godot and the vast majority of other applications is that +X is east and -X is west. Due to Godot's right-handed Y-is-up coordinate system, this implies that +Z is south and -Z is north. In Blender, this means that +Y is north and -Y is south.

Exporting textures separately

While textures can be exported with a model in certain file formats, such as glTF 2.0, you can also export them separately. Godot uses PBR (physically based rendering) for its materials, so if a texturing program can export PBR textures, they can work in Godot. This includes the Substance suite [https://www.adobe.com/creativecloud/3d-ar.html], ArmorPaint (open source) [https://armorpaint.org/], and Material Maker (open source) [https://github.com/RodZill4/material-maker].

See also

For more information on Godot's materials, see Standard Material 3D and ORM Material 3D.

Exporting considerations

Since GPUs can only render triangles, meshes that contain quads or N-gons have to be *triangulated* before they can be rendered. Godot can triangulate meshes on import, but results may be unpredictable or incorrect, especially with N-gons. Regardless of the target application, triangulating *before* exporting the scene will lead to more consistent results and should be done whenever possible.

To avoid issues with incorrect triangulation after importing in Godot, it is recommended to make the 3D modeling software triangulate objects on its own. In Blender, this can be done by adding a Triangulate modifier to your objects and making sure **Apply Modifiers** is checked in the export dialog. Alternatively, depending on the exporter, you may be able to find and enable a **Triangulate Faces** option in the export dialog.

To avoid issues with 3D selection in the editor, it is recommended to apply the object transform in the 3D modeling software before exporting the scene.

Note

It is important that the mesh is not deformed by bones when exporting. Make sure that the skeleton is reset to its T-pose or default rest pose before exporting with your favorite 3D editor.

Lighting considerations

While it's possible to import lights from a 3D scene using the glTF, `.blend` or Collada formats, it's generally advised to design the scene's lighting in the Godot editor after importing the scene.

This allows you to get a more accurate feel for the final result, as different engines will render lights in a different manner. This also avoids any issues with lights appearing excessively strong or faint as a result of the import process.

Node type customization using name suffixes

Many times, when editing a scene, there are common tasks that need to be done after exporting:

- Adding collision detection to objects.
- Setting objects as navigation meshes.
- Deleting nodes that are not used in the game engine (like specific lights used for modeling).

To simplify this workflow, Godot offers several suffixes that can be added to the names of the objects in your 3D modeling software. When imported, Godot will detect suffixes in object names and will perform actions automatically.

Warning

All the suffixes described below can be used with `-`, `$`, and `_` and are **case-insensitive**.

Remove nodes (-noimp)

Objects that have the `-noimp` suffix will be removed at import-time no matter what their type is. They will not appear in the imported scene.

This is equivalent to enabling **Skip Import** for a node in the Advanced Import Settings dialog.

Create collisions (-col, -convcol, -colonly, -convcolonly)

The option `-col` will work only for Mesh objects. If it is detected, a child static collision node will be added, using the same geometry as the mesh. This will create a triangle mesh collision shape, which is a slow, but accurate option for collision detection. This option is usually what you want for level geometry (but see also `-colonly` below).

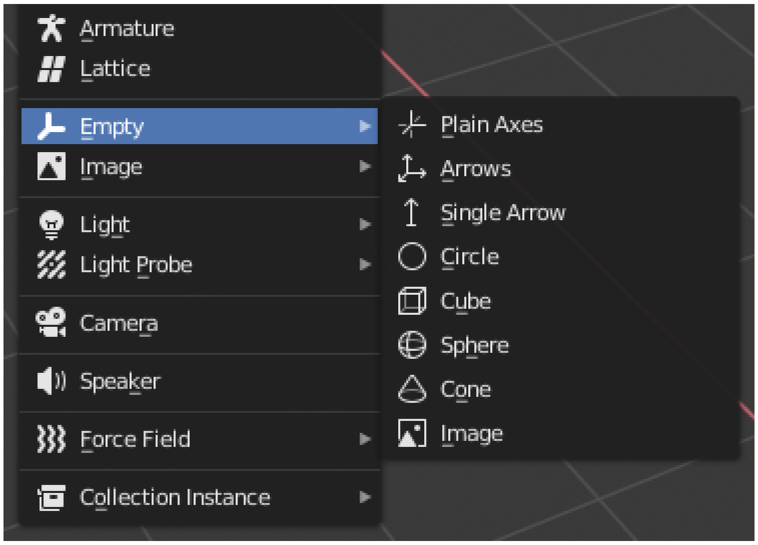
The option `-convcol` will create a ConvexPolygonShape3D instead of a ConcavePolygonShape3D. Unlike triangle meshes which can be concave, a convex shape can only accurately represent a shape that doesn't have any concave angles (a pyramid is convex, but a hollow box is concave). Due to this, convex collision shapes are generally not suited for level geometry. When representing simple enough meshes, convex collision shapes can result in better performance compared to a triangle collision shape. This option is ideal for simple or dynamic objects that require mostly-accurate collision detection.

However, in both cases, the visual geometry may be too complex or not smooth enough for collisions. This can create physics glitches and slow down the engine unnecessarily.

To solve this, the `-colonly` modifier exists. It will remove the mesh upon importing and will create a StaticBody3D collision instead. This helps the visual mesh and actual collision to be separated.

The option `-convcolonly` works in a similar way, but will create a ConvexPolygonShape3D instead using convex decomposition.

With Collada files, the option `-colonly` can also be used with Blender's empty objects. On import, it will create a StaticBody3D with a collision node as a child. The collision node will have one of a number of predefined shapes, depending on Blender's empty draw type:


Armature
Lattice
Empty
Image
Light
Light Probe
Camera
Speaker
Force Field
Collection Instance
Plain Axes
Arrows
Single Arrow
Circle
Cube
Sphere
Cone
Image

Choosing a draw type for an Empty on creation in Blender

- Single arrow will create a SeparationRayShape3D.
- Cube will create a BoxShape3D.
- Image will create a WorldBoundaryShape3D.
- Sphere (and the others not listed) will create a SphereShape3D.

When possible, **try to use a few primitive collision shapes** instead of triangle mesh or convex shapes. Primitive shapes often have the best performance and reliability.

Note

For better visibility on Blender's editor, you can set the "X-Ray" option on collision empties and set some distinct color for them by changing **Edit > Preferences > Themes > 3D Viewport > Empty**.

If using Blender 2.79 or older, follow these steps instead: **User Preferences > Themes > 3D View > Empty**.

See also

See Collision shapes (3D) for a comprehensive overview of collision shapes.

Create Occluder (-occ, -occonly)

If a mesh is imported with the `-occ` suffix an Occluder3D node will be created based on the geometry of the mesh, it does not replace the mesh. A mesh node with the `-occonly` suffix will be converted to an Occluder3D on import.

Create navigation (-navmesh)

A mesh node with the `-navmesh` suffix will be converted to a navigation mesh. The original Mesh object will be removed at import-time.

Create a VehicleBody (-vehicle)

A mesh node with the `-vehicle` suffix will be imported as a child to a VehicleBody3D node.

Create a VehicleWheel (-wheel)

A mesh node with the `-wheel` suffix will be imported as a child to a VehicleWheel3D node.

Rigid Body (-rigid)

A mesh node with the `-rigid` suffix will be imported as a RigidBody3D.

Animation loop (-loop, -cycle)

Animation clips in the source 3D file that start or end with the token `loop` or `cycle` will be imported as a Godot Animation with the loop flag set. **Unlike the other suffixes described above, this does not require a hyphen.**

In Blender, this requires using the NLA Editor and naming the Action with the `loop` or `cycle` prefix or suffix.

Import configuration

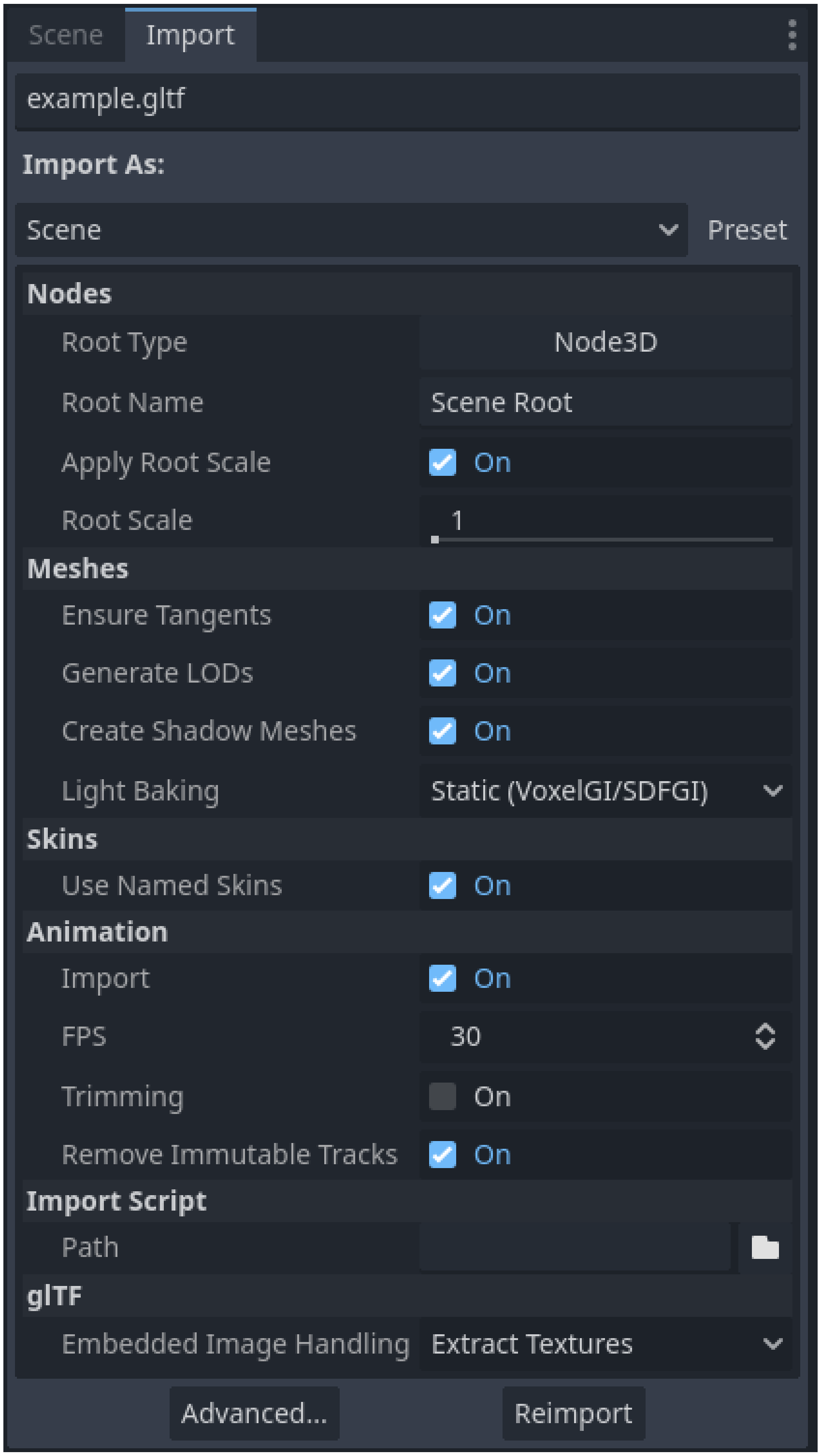
Godot provides several ways to customize the imported data, such as the import dock, the advanced import setting dialog, and inherited scenes. This can be used to make further changes to the imported scene, such as adjusting meshes, adding physics information, and adding new nodes. You can also write a script that runs code at the end of the import process to perform arbitrary customization.

Note that, when applicable, modifying the original data should be preferred to configuring the scene after import. This helps minimize the differences between the 3D modeling application and the imported scene. See the Model export considerations and Node type customization using name suffixes articles for more information.

Import workflows

Since Godot can only save its own scene format (`.tscn`/`.scn`), Godot cannot save over the original 3D scene file (which uses a different format). This is also a safer approach as it avoids making accidental changes to the source file.

To allow customizing the scene and its materials, Godot's scene importer allows for different workflows regarding how data is imported.



example.gltf

Import As:

Scene

Nodes

Root Type: Node3D

Root Name: Scene Root

Apply Root Scale: ☑ On

Root Scale: 1

Meshes

Ensure Tangents: ☑ On

Generate LODs: ☑ On

Create Shadow Meshes: ☑ On

Light Baking: Static (VoxelGI/SDFGI)

Skins

Use Named Skins: ☑ On

Animation

Import: ☑ On

FPS: 30

Trimming: ☐ On

Remove Immutable Tracks: ☑ On

Import Script

Path:

glTF

Embedded Image Handling: Extract Textures

Import dock after selecting a 3D scene in the FileSystem dock

This import process is customizable using 3 separate interfaces, depending on your needs:

- The **Import** dock, after selecting the 3D scene by clicking it once in the FileSystem dock.
- The **Advanced Import Settings** dialog, which can be accessed by double-clicking the 3D scene in the FileSystem dock or by clicking the **Advanced…** button in the Import dock. This allows you to customize per-object options in Godot.
- Import hints, which are special suffixes added to object names in the 3D modeling software. This allows you to customize per-object options in the 3D modeling software.

For basic customization, using the Import dock suffices. However, for more complex operations such as defining material overrides on a per-material basis, you'll need to use the Advanced Import Settings dialog, import hints, or possibly both.

Using the Import dock

The following options can be adjusted in the Import dock after selecting a 3D scene in the FileSystem dock:

- **Root Type:** The node type to use as a root node. Using node types that inherit from Node3D is recommended. Otherwise, you'll lose the ability to position the node directly in the 3D editor.
- **Root Name:** The name of the root node in the imported scene. This is generally not noticeable when instancing the scene in the editor (or drag-and-dropping from the FileSystem dock), as the root node is renamed to match the filename in this case.
- **Apply Root Scale:** If enabled, **Root Scale** will be *applied* on the meshes and animations directly, while keeping the root node's scale to the default *(1, 1, 1)*. This means that if you add a child node later on within the imported scene, it won't be

scaled. If disabled, **Root Scale** will multiply the scale of the root node instead.

Meshes

- **Ensure Tangents:** If checked, generate vertex tangents using Mikktspace [http://www.mikktspace.com/] if the input meshes don't have tangent data. When possible, it's recommended to let the 3D modeling software generate tangents on export instead on relying on this option. Tangents are required for correct display of normal and height maps, along with any material/shader features that require tangents. If you don't need material features that require tangents, disabling this can reduce output file size and speed up importing if the source 3D file doesn't contain tangents.
- **Generate LODs:** If checked, generates lower detail variants of the mesh which will be displayed in the distance to improve rendering performance. Not all meshes benefit from LOD, especially if they are never rendered from far away. Disabling this can reduce output file size and speed up importing. See Mesh level of detail (LOD) for more information.
- **Create Shadow Meshes:** If checked, enables the generation of shadow meshes on import. This optimizes shadow rendering without reducing quality by welding vertices together when possible. This in turn reduces the memory bandwidth required to render shadows. Shadow mesh generation currently doesn't support using a lower detail level than the source mesh (but shadow rendering will make use of LODs when relevant).
- **Light Baking:** Configures the meshes' global illumination mode in the 3D scene. If set to **Static Lightmaps**, sets the meshes' GI mode to **Static** and generates UV2 on import for lightmap baking.
- **Lightmap Texel Size:** Only visible if **Light Baking** is set to **Static Lightmaps**. Controls the size of each texel on the baked lightmap. A smaller value results in more precise lightmaps, at the cost of larger lightmap sizes and longer bake times.

Skins

- **Use Named Skins:** If checked, use named Skins for animation. The MeshInstance3D node contains 3 properties of relevance here: a skeleton NodePath pointing to the Skeleton3D node (usually `..`), a mesh, and a skin:
    - The Skeleton3D node contains a list of bones with names, their pose and rest, a name and a parent bone.
    - The mesh is all of the raw vertex data needed to display a mesh. In terms of the mesh, it knows how vertices are weight-painted and uses some internal numbering often imported from 3D modeling software.
    - The skin contains the information necessary to bind this mesh onto this Skeleton3D. For every one of the internal bone IDs chosen by the 3D modeling software, it contains two things. Firstly, a Matrix known as the Bind Pose Matrix, Inverse Bind Matrix, or IBM for short. Secondly, the Skin contains each bone's name (if **Use Named Skins** is enabled), or the bone's index within the Skeleton3D list (if **Use Named Skins** is disabled).

Together, this information is enough to tell Godot how to use the bone poses in the Skeleton3D node to render the mesh from each MeshInstance3D. Note that each MeshInstance3D may share binds, as is common in models exported from Blender, or each MeshInstance3D may use a separate Skin object, as is common in models exported from other tools such as Maya.

Animation

- **Import:** If checked, import animations from the 3D scene.
- **FPS:** The number of frames per second to use for baking animation curves to a series of points with linear interpolation. It's recommended to configure this value to match the value you're using as a baseline in your 3D modeling software. Higher values result in more precise animation with fast movement

changes, at the cost of higher file sizes and memory usage. Thanks to interpolation, there is usually not much benefit in going above 30 FPS (as the animation will still appear smooth at higher rendering framerates).
- **Trimming:** Trim the beginning and end of animations if there are no keyframe changes. This can reduce output file size and memory usage with certain 3D scenes, depending on the contents of their animation tracks.
- **Remove Immutable Tracks:** Remove animation tracks that only contain default values. This can reduce output file size and memory usage with certain 3D scenes, depending on the contents of their animation tracks.

Import Script

- **Path:** Path to an import script, which can run code *after* the import process has completed for custom processing. See Using import scripts for automation for more information.

glTF

- **Embedded Texture Handling:** Controls how textures embedded within glTF scenes should be handled. **Discard All Textures** will not import any textures, which is useful if you wish to manually set up materials in Godot instead. **Extract Textures** extracts textures to external images, resulting in smaller file sizes and more control over import options. **Embed as Basis Universal** and **Embed as Uncompressed** keeps the textures embedded in the imported scene, with and without VRAM compression respectively.

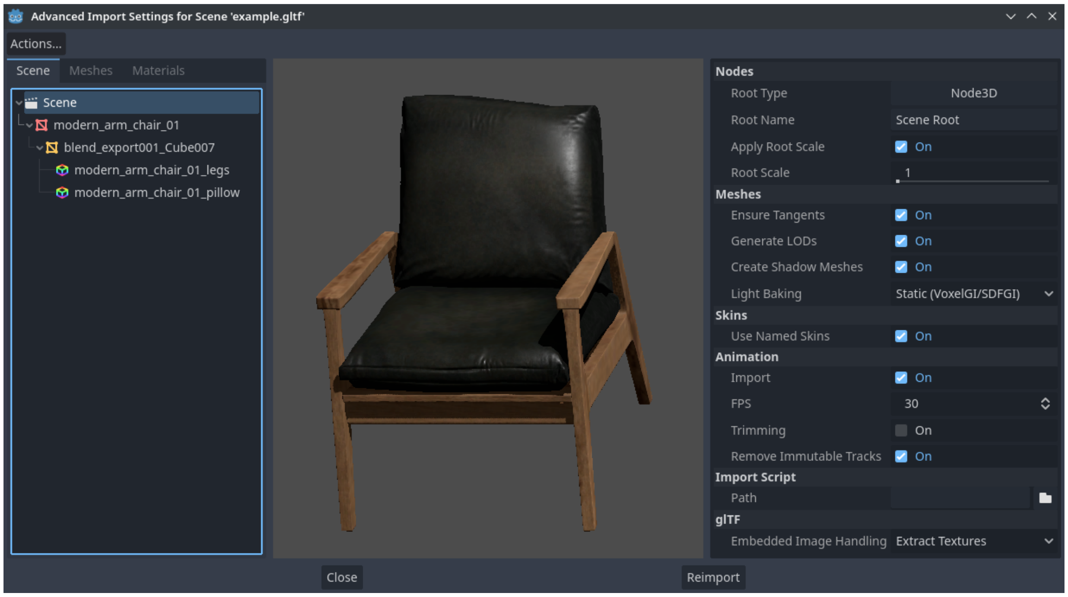
FBX

- **Importer** Which import method is used. ubfx handles fbx files as fbx files. FBX2glTF converts FBX files to glTF on import and requires additional setup. FBX2glTF is not recommended unless

you have a specific rason to use it over ufbx or working with a different file format.
- **Allow Geometry Helper Nodes** enables or disables geometry helper nodes
- **Embedded Texture Handling:** Controls how textures embedded within fbx scenes should be handled. **Discard All Textures** will not import any textures, which is useful if you wish to manually set up materials in Godot instead. **Extract Textures** extracts textures to external images, resulting in smaller file sizes and more control over import options. **Embed as Basis Universal** and **Embed as Uncompressed** keeps the textures embedded in the imported scene, with and without VRAM compression respectively.

Using the Advanced Import Settings dialog

The first tab you'll see is the **Scene** tab. The options available in the panel on the right are identical to the Import dock, but you have access to a 3D preview. The 3D preview can be rotated by holding down the left mouse button then dragging the mouse. Zoom can be adjusted using the mouse wheel.



Advanced Import Settings for Scene 'example.gltf'

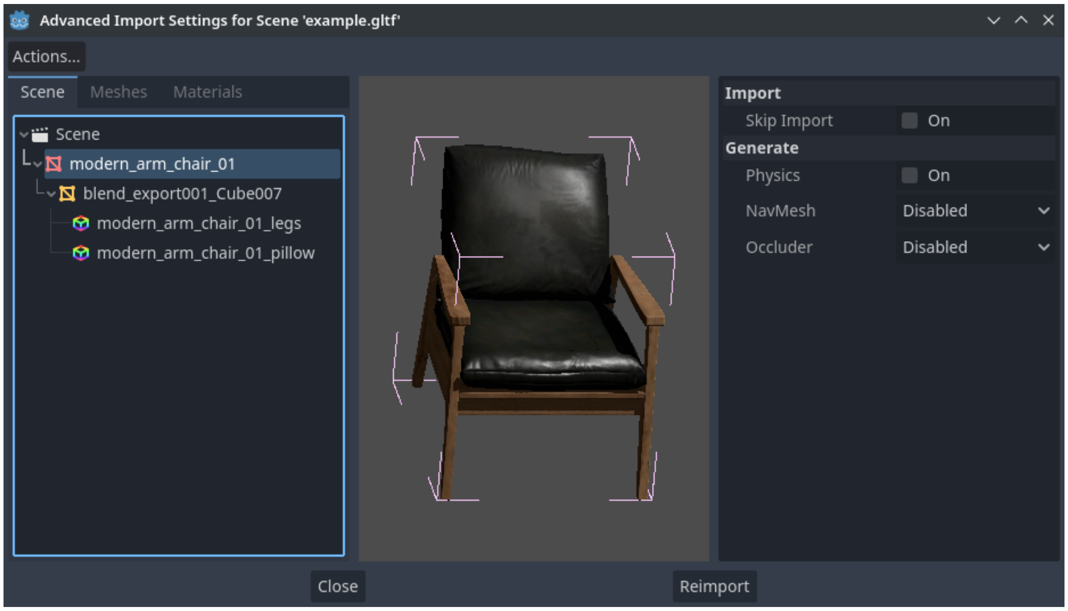
- Scene
  - modern_arm_chair_01
    - blend_export001_Cube007
      - modern_arm_chair_01_legs
      - modern_arm_chair_01_pillow

Nodes	
Root Type	Node3D
Root Name	Scene Root
Apply Root Scale	☑ On
Root Scale	1
Meshes	
Ensure Tangents	☑ On
Generate LODs	☑ On
Create Shadow Meshes	☑ On
Light Baking	Static (VoxelGI/SDFGI)
Skins	
Use Named Skins	☑ On
Animation	
Import	☑ On
FPS	30
Trimming	☐ On
Remove Immutable Tracks	☑ On
Import Script	
Path	
glTF	
Embedded Image Handling	Extract Textures

Advanced Import Settings dialog (Scene tab). Credit: [Modern Arm Chair 01 - Poly Haven](https://polyhaven.com/a/modern_arm_chair_01) [https://polyhaven.com/a/modern_arm_chair_01]

Configuring node import options

You can select individual nodes that compose the scene while in the **Scene** tab using the tree view at the left:


Advanced Import Settings for Scene 'example.gltf'
Actions...
Scene
Meshes
Materials
Scene
modern_arm_chair_01
blend_export001_Cube007
modern_arm_chair_01_legs
modern_arm_chair_01_pillow
Import
Generate
Close
Reimport

Skip Import ☐ On

Physics ☐ On

NavMesh Disabled

Occluder Disabled

Selecting a node in the Advanced Import Settings dialog (Materials tab)

This exposes several per-node import options:

- **Skip Import:** If checked, the node will not be present in the final imported scene. Enabling this disables all other options.
- **Generate > Physics:** If checked, generates a PhysicsBody3D *parent* node with collision shapes that are *siblings* to the MeshInstance3D node.
- **Generate > NavMesh:** If checked, generates a NavigationRegion3D *child* node for navigation. **Mesh + NavMesh** will keep the original mesh visible, while **NavMesh Only** will only import the navigation mesh (without a visual

representation). **NavMesh Only** is meant to be used when you've manually authored a simplified mesh for navigation.
- **Generate > Occluder:** If checked, generates an OccluderInstance3D *sibling* node for occlusion culling using the mesh's geometry as a basis for the occluder's shape. **Mesh + Occluder** will keep the original mesh visible, while **Occluder Only** will only import the occluder (without a visual representation). **Occluder Only** is meant to be used when you've manually authored a simplified mesh for occlusion culling.

These options are only visible if some of the above options are enabled:

- **Physics > Body Type:** Only visible if **Generate > Physics** is enabled. Controls the PhysicsBody3D that should be created. **Static** creates a StaticBody3D, **Dynamic** creates a RigidBody3D, **Area** creates an Area3D.
- **Physics > Shape Type:** Only visible if **Generate > Physics** is enabled. **Trimesh** allows for precise per-triangle collision, but it can only be used with a **Static** body type. Other types are less precise and may require manual configuration, but can be used with any body type. For static level geometry, use **Trimesh**. For dynamic geometry, use primitive shapes if possible for better performance, or use one of the convex decomposition modes if the shape is large and complex.
- **Decomposition > Advanced:** Only visible if **Physics > Shape Type** is **Decompose Convex**. If checked, allows adjusting advanced decomposition options. If disabled, only a preset **Precision** can be adjusted (which is usually sufficient).
- **Decomposition > Precision:** Only visible if **Physics > Shape Type** is **Decompose Convex**. Controls the precision to use for convex decomposition. Higher values result in more detailed collision, at the cost of slower generation and increased CPU usage during physics simulation. To improve performance,

it's recommended to keep this value as low as possible for your use cases.

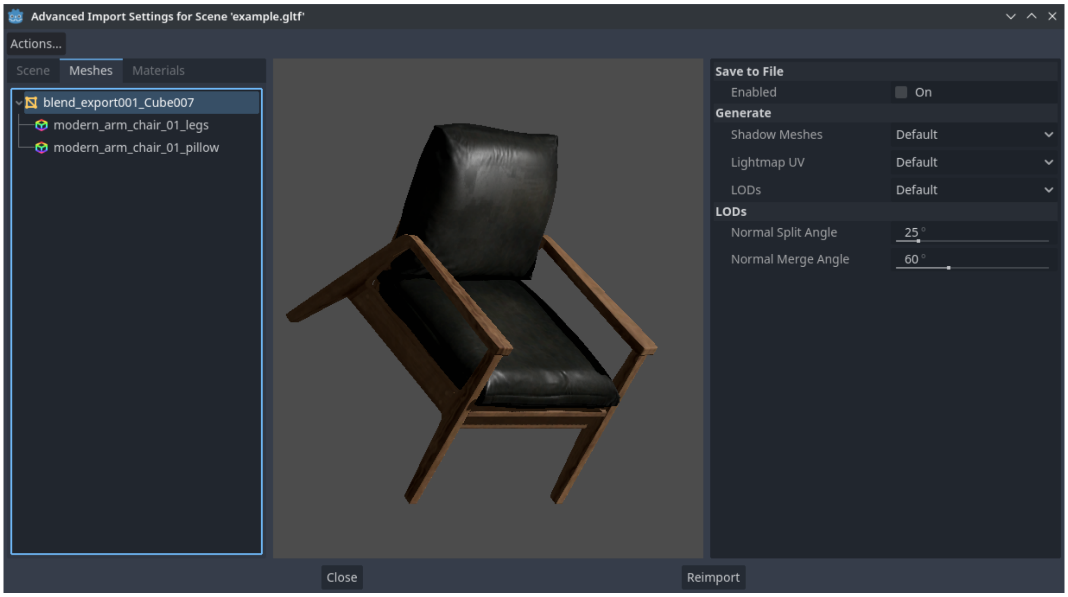
- **Occluder > Simplification Distance:** Only visible if **Generate > Occluder** is set to **Mesh + Occluder** or **Occluder Only**. Higher values result in an occluder mesh with fewer vertices (resulting in decreased CPU utilization), at the cost of more occlusion culling issues (such as false positives or false negatives). If you run into objects disappearing when they shouldn't when the camera is near a certain mesh, try decreasing this value.

Configuring mesh and material import options

In the Advanced Import Settings dialog, there are 2 ways to select individual meshes or materials:

- Switch to the **Meshes** or **Materials** tab in the top-left corner of the dialog.
- Stay in the **Scene** tab, but unfold the options on the tree view on the left. After choosing a mesh or material, this presents the same information as the **Meshes** and **Materials** tabs, but in a tree view instead of a list.

If you select a mesh, different options will appear in the panel on the right:



blend_export001_Cube007
modern_arm_chair_01_legs
modern_arm_chair_01_pillow

Save to File
Enabled ☐ On
Generate
Shadow Meshes Default
Lightmap UV Default
LODs Default
LODs
Normal Split Angle 25 °
Normal Merge Angle 60 °

Advanced Import Settings dialog (Meshes tab)

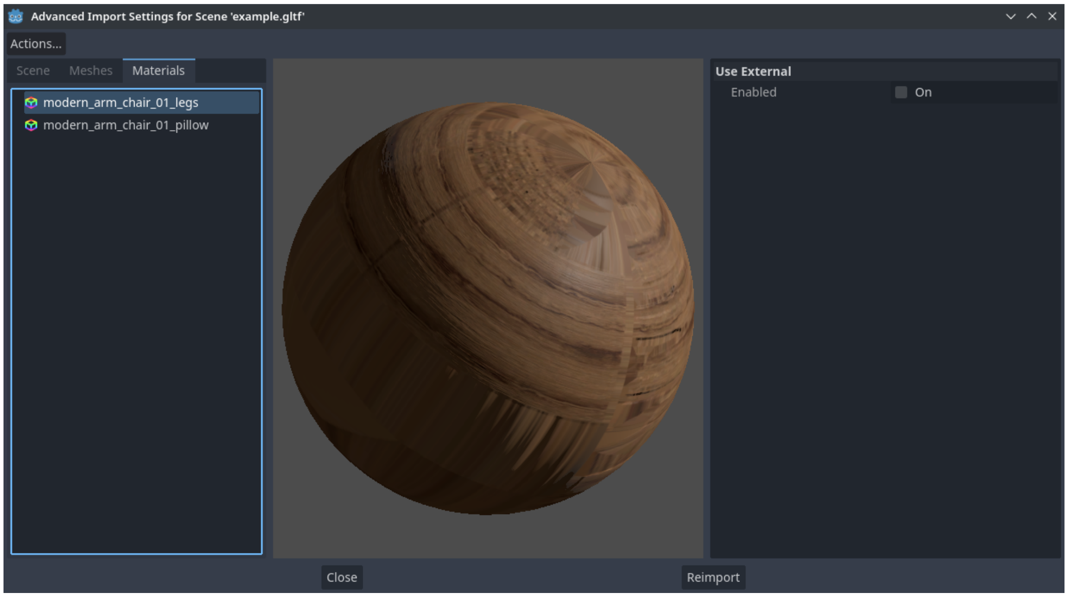
The options are as follows:

- **Save to File:** Saves the Mesh *resource* to an external file (this isn't a scene file). You generally don't need to use this for placing the mesh in a 3D scene – instead, you should instance the 3D scene directly. However, having direct access to the Mesh resource is useful for specific nodes, such as MeshInstance3D, MultiMeshInstance3D, GPUParticles3D or CPUParticles3D. - You will also need to specify an output file path using the option that appears after enabling **Save to File**. It's recommended to use the `.res` output file extension for smaller file sizes and faster loading speeds, as `.tres` is inefficient for writing large amounts of data.
- **Generate > Shadow Meshes:** Per-mesh override for the **Meshes > Create Shadow Meshes** scene-wide import option described in Using the Import dock. **Default** will use the scene-wide import option, while **Enable** or **Disable** can forcibly enable or disable this behavior on a specific mesh.
- **Generate > Lightmap UV:** Per-mesh override for the **Meshes > Light Baking** scene-wide import option described in Using

the Import dock. **Default** will use the scene-wide import option, while **Enable** or **Disable** can forcibly enable or disable this behavior on a specific mesh. - Setting this to **Enable** on a scene with the **Static** light baking mode is equivalent to configuring this mesh to use **Static Lightmaps**. Setting this to **Disable** on a scene with the **Static Lightmaps** light baking mode is equivalent to configuring this mesh to use **Static** instead.

- **Generate > LODs:** Per-mesh override for the **Meshes > Generate LODs** scene-wide import option described in Using the Import dock. **Default** will use the scene-wide import option, while **Enable** or **Disable** can forcibly enable or disable this behavior on a specific mesh.
- **LODs > Normal Split Angle:** The minimum angle difference between two vertices required to preserve a geometry edge in mesh LOD generation. If running into visual issues with LOD generation, decreasing this value may help (at the cost of less efficient LOD generation).
- **LODs > Normal Merge Angle:** The minimum angle difference between two vertices required to preserve a geometry edge in mesh LOD generation. If running into visual issues with LOD generation, decreasing this value may help (at the cost of less efficient LOD generation).

If you select a material, only one option will appear in the panel on the right:


Advanced Import Settings for Scene 'example.gltf'
Actions...
Scene
Meshes
Materials
modern_arm_chair_01_legs
modern_arm_chair_01_pillow
Close
Reimport

Use External
Enabled ☐ On

Advanced Import Settings dialog (Materials tab)

When **Use External** is checked and an output path is specified, this lets you use an external material instead of the material that is included in the original 3D scene file; see the section below.

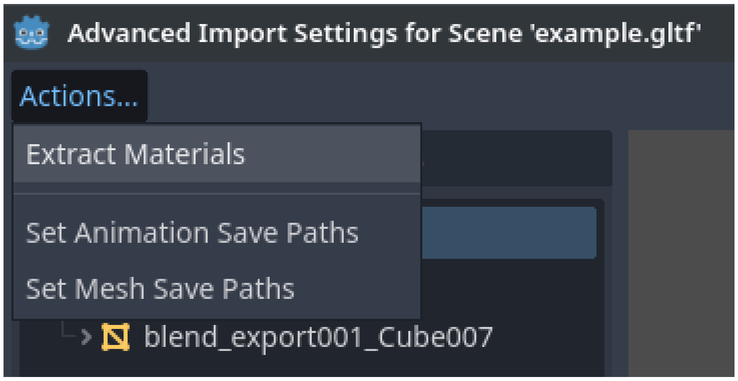
Extracting materials to separate files

While Godot can import materials authored in 3D modeling software, the default configuration may not be suitable for your needs. For example:

- You want to configure material features not supported by your 3D application.
- You want to use a different texture filtering mode, as this option is configured in the material since Godot 4.0 (and not in the image).
- You want to replace one of the materials with an entirely different material, such as a custom shader.

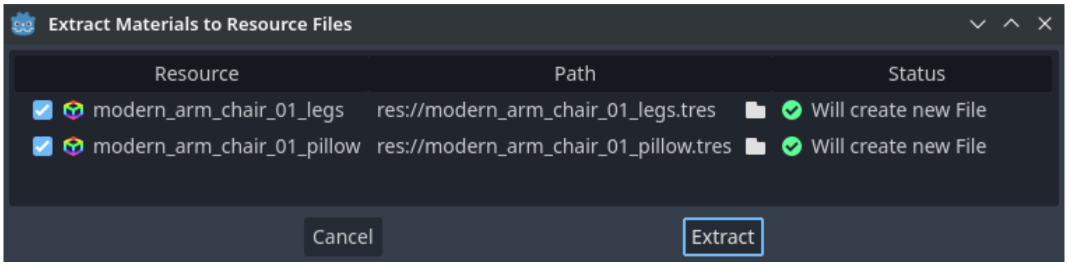
To be able to modify the 3D scene's materials in the Godot editor, you need to use *external* material resources.

In the top-left corner of the Advanced Import Settings dialog, choose **Actions... > Extract Materials**:


Advanced Import Settings for Scene 'example.gltf'
Actions...
Extract Materials
Set Animation Save Paths
Set Mesh Save Paths
blend_export001_Cube007

Extracting all built-in materials to external resources in the Advanced Import Settings dialog

After choosing this option, select a folder to extract material `.tres` files to, then confirm the extraction:


Extract Materials to Resource Files
Cancel
Extract

Resource	Path	Status
☑ modern_arm_chair_01_legs	res://modern_arm_chair_01_legs.tres	✓ Will create new File
☑ modern_arm_chair_01_pillow	res://modern_arm_chair_01_pillow.tres	✓ Will create new File

Confirming material extraction in the Advanced Import Settings subdialog

Note

After extracting materials, the 3D scene will automatically be configured to use external material references. As a result, you don't need to manually enable **Use External** on every material to make the external `.tres` material effective.

When **Use External** is enabled, remember that the Advanced Import Settings dialog will keep displaying the mesh's original materials (the ones designed in the 3D modeling software). This means your customizations to the materials won't be visible within this dialog. To preview your modified materials, you need to place the imported 3D scene in another scene using the editor.

Godot will not overwrite changes made to extracted materials when the source 3D scene is reimported. However, if the material name is changed in the source 3D file, the link between the original material and the extracted material will be lost. As a result, you'll need to use the Advanced Import Settings dialog to associate the renamed material to the existing extracted material.

The above can be done in the dialog's **Materials** tab by selecting the material, enabling **Save to File**, then specifying the save path using the **Path** option that appears after enabling **Save to File**.

Using import scripts for automation

A special script to process the whole scene after import can be provided. This is great for post-processing, changing materials, doing funny stuff with the geometry, and more.

Create a script that is not attached to any node by right-clicking in the FileSystem dock and choosing **New > Script...**. In the script editor, write the following:

```
@tool # Needed so it runs in editor.
extends EditorScenePostImport

# This sample changes all node names.
# Called right after the scene is imported and gets the root
node.
func _post_import(scene):
    # Change all node names to "modified_[oldnodename]"
    iterate(scene)
    return scene # Remember to return the imported scene

# Recursive function that is called on every node
# (for demonstration purposes; EditorScenePostImport only
requires a `_post_import(scene)` function).
func iterate(node):
    if node != null:
        print_rich("Post-import: [b]%s[/b] -> [b]%s[/b]" %
[node.name, "modified_" + node.name])
        node.name = "modified_" + node.name
        for child in node.get_children():
            iterate(child)
```

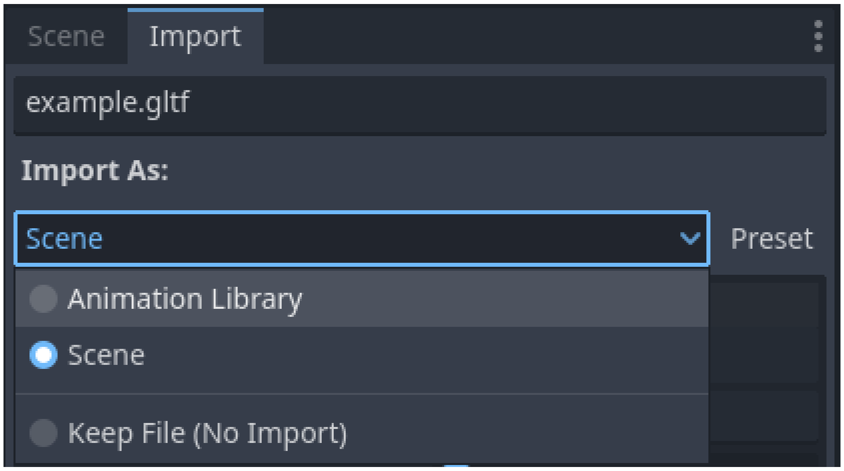
The `_post_import(scene: Node)` function takes the imported scene as argument (the parameter is actually the root node of the scene). The scene that will finally be used **must** be returned (even if the scene can be entirely different).

To use your script, locate the script in the import tab's "Path" option under the "Import Script" category.

Using animation libraries

As of Godot 4.0, you can choose to import **only** animations from a glTF file and nothing else. This is used in some asset pipelines to distribute animations separately from models. For example, this allows you to use one set of animations for several characters, without having to duplicate animation data in every character.

To do so, select the glTF file in the FileSystem dock, then change the import mode to Animation Library in the Import dock:



Changing the import type to Animation Library in the Import dock

Click **Reimport** and restart the editor when prompted. After restarting, the glTF file will be imported as an AnimationLibrary instead of a PackedScene. This animation library can then be referenced in an AnimationPlayer node.

The import options that are visible after changing the import mode to Animation Library act the same as when using the Scene import mode. See Using the Import dock for more information.

Filter script

It is possible to specify a filter script in a special syntax to decide which tracks from which animations should be kept.

The filter script is executed against each imported animation. The syntax consists of two types of statements, the first for choosing which animations to filter, and the second for filtering individual tracks within the matched animation. All name patterns are performed using a case-insensitive expression match, with support for `?` and `*` wildcards (using String.matchn() under the hood).

The script must start with an animation filter statement (as denoted by the line beginning with an `@`). For example, if we would like to apply filters to all imported animations which have a name ending in `"_Loop"`:

```
@+*_Loop
```

Similarly, additional patterns can be added to the same line, separated by commas. Here is a modified example to additionally *include* all animations with names that begin with `"Arm_Left"`, but also *exclude* all animations which have names ending in `"Attack"`:

```
@+*_Loop, +Arm_Left*, -*Attack
```

Following the animation selection filter statement, we add track filtering patterns to indicate which animation tracks should be kept or discarded. If no track filter patterns are specified, then all tracks within the matched animations will be discarded!

It's important to note that track filter statements are applied in order for each track within the animation, this means that one line may include a track, a later rule can still discard it. Similarly, a track excluded by an early rule may then be re-included once again by a filter rule further down in the filter script.

For example: include all tracks in animations with names ending in `"_Loop"`, but discard any tracks affecting a `"Skeleton"` which end in `"Control"`, unless they have `"Arm"` in their name:

```
@+*_Loop
+*
-Skeleton:*Control
+*Arm*
```

In the above example, tracks like `"Skeleton:Leg_Control"` would be discarded, while tracks such as `"Skeleton:Head"` or `"Skeleton:Arm_Left_Control"` would be retained.

Any track filter lines that do not begin with a `+` or `-` are ignored.

Storage

By default, animations are saved as built-in. It is possible to save them to a file instead. This allows adding custom tracks to the animations and keeping them after a reimport.

Optimizer

When animations are imported, an optimizer is run, which reduces the size of the animation considerably. In general, this should always be turned on unless you suspect that an animation might be broken due to it being enabled.

Clips

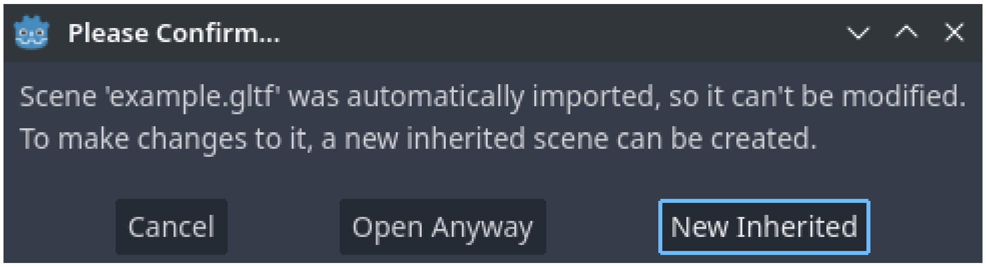
It is possible to specify multiple animations from a single timeline as clips. For this to work, the model must have only one animation that is named `default`. To create clips, change the clip amount to something greater than zero. You can then name a clip, specify which frames it starts and stops on, and choose whether the animation loops or not.

Scene inheritance

In many cases, it may be desired to make manual modifications to the imported scene. By default, this is not possible because if the source 3D asset changes, Godot will re-import the *whole* scene.

However, it is possible to make local modifications by using *scene inheritance*. If you try to open the imported scene using **Scene >**

Open Scene... or **Scene > Quick Open Scene...**, the following dialog will appear:



Scene 'example.gltf' was automatically imported, so it can't be modified.
To make changes to it, a new inherited scene can be created.

Dialog when opening an imported 3D scene in the editor

In inherited scenes, the only limitations for modification are:

- Nodes from the base scene can't be removed, but additional nodes can be added anywhere.
- Subresources can't be edited. Instead, you need to save them externally as described above.

Other than that, everything is allowed.

Retargeting 3D Skeletons

To share animations among multiple Skeletons

Godot has Position/Rotation/Scale 3D tracks (which this document calls "Transform" tracks) with Nodepaths to bones for Skeleton bone animation. This means you can't share animations between multiple Skeletons just by using the same bone names.

Godot allows each bone to have a parent-child relationship and can have rotation and scale as well as position, which means that bones that share a name can still have different Transform values.

The Skeleton stores the Transform values necessary for the default pose as Bone Rest. If Bone Pose is equal to Bone Rest, it means that the Skeleton is in the default pose.

Note

Godot 3.x and Godot 4.0+ have different Bone Pose behaviors. In Godot 3.x, Bone Pose is relative to Bone Rest, but in Godot 4.0+, it includes Bone Rest. See this article [https://godotengine.org/article/animation-data-redesign-40].

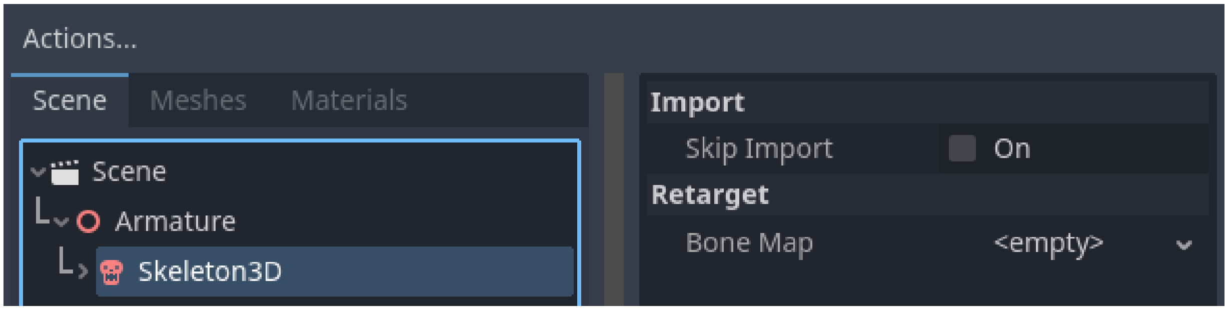
Skeletal models have different Bone Rests depending on the environment from which they were exported. For example, the bones of a glTF model output from Blender have "Edit Bone Orientation" as the Bone Rest rotation. However, there are skeletal models without any Bone Rest rotations, such as the glTF model output from Maya.

To share animations in Godot, it is necessary to match Bone Rests as well as Bone Names to remove unwanted tracks in some cases. In Godot 4.0+, you can do that using the scene importer.

Options for Retargeting

Bone Map

When you select the Skeleton3D node in the advanced scene import menu, a menu will appear on the right-hand side containing the "Retarget" section. The Retarget section has a single property `bone_map`.



Skip Import ☐ On

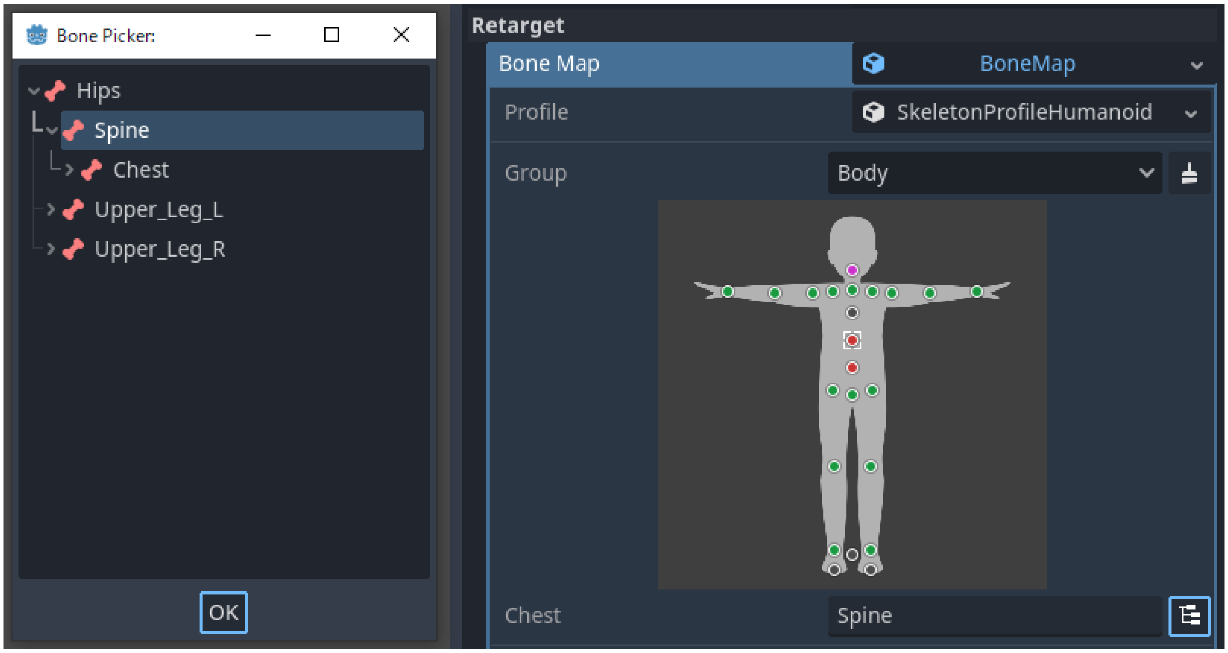
Bone Map <empty>

With the Skeleton node selected, first set up a new BoneMap and SkeletonProfile. Godot has a preset called SkeletonProfileHumanoid for humanoid models. This tutorial proceeds with the assumption that you are using SkeletonProfileHumanoid.

Note

If you need a profile that is different from SkeletonProfileHumanoid, you can export a SkeletonProfile from the editor by selecting a Skeleton3D and using the **Skeleton3D** menu in the 3D viewport's toolbar.

When you use SkeletonProfileHumanoid, auto-mapping will be performed when the SkeletonProfile is set. If the auto-mapping does not work well, you can map bones manually.


Bone Map: BoneMap

Profile: SkeletonProfileHumanoid

Group: Body

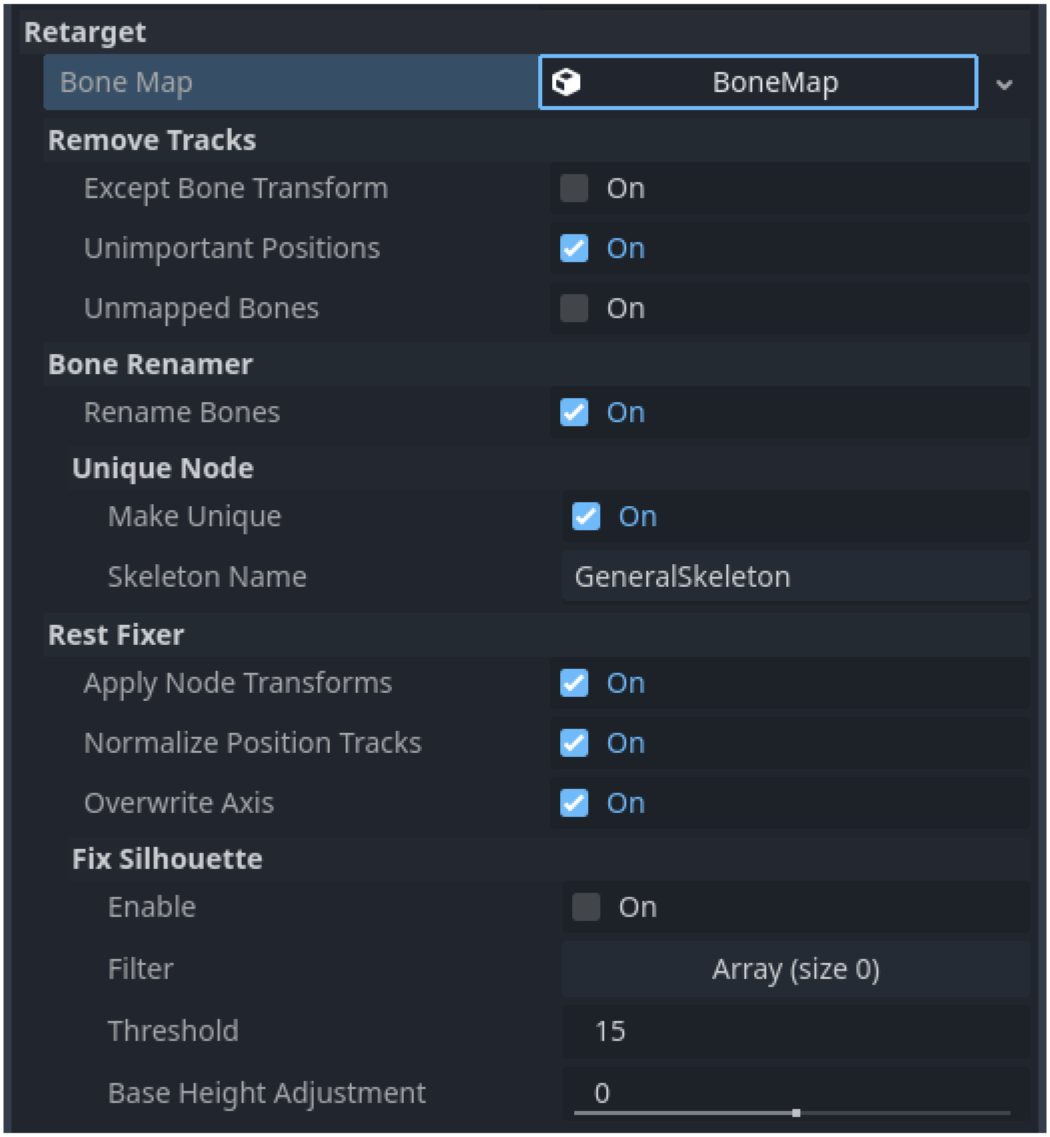
Chest: Spine

Any missing, duplicate or incorrect parent-child relationship mappings will be indicated by a magenta / red button (depending on the editor setting). It does not block the import process, but it warns that animations may not be shared correctly.

Note

The auto-mapping uses pattern matching for the bone names. So we recommend to use common English names for bones.

After you set up the `bone_map`, several options are available in the sections below.



Retarget

Bone Map: BoneMap

Remove Tracks

Except Bone Transform ☐ On

Unimportant Positions ☑ On

Unmapped Bones ☐ On

Bone Renamer

Rename Bones ☑ On

Unique Node

Make Unique ☑ On

Skeleton Name: GeneralSkeleton

Rest Fixer

Apply Node Transforms ☑ On

Normalize Position Tracks ☑ On

Overwrite Axis ☑ On

Fix Silhouette

Enable ☐ On

Filter: Array (size 0)

Threshold: 15

Base Height Adjustment: 0

Remove Tracks

If you import resources as an AnimationLibrary that will be shared, we recommend to enable these options. However, if you import resources as scenes, these should be disabled in some cases. For example, if you import a character with animated accessories, these options may cause the accessories to not animate.

Except Bone Transform

Removes any tracks except the bone Transform track from the animations.

Unimportant Positions

Removes Position tracks other than `root_bone` and `scale_base_bone` defined in SkeletonProfile from the animations. In SkeletonProfileHumanoid, this means that to remove Position tracks other than "Root" and "Hips". Since Godot 4.0+, animations include Bone Rest in the Transform value. If you disable this option, the animation may change the body shape unpredictably.

Unmapped Bones

Removes unmapped bone Transform tracks from the animations.

Bone Renamer

Rename Bones

Rename the mapped bones.

Unique Node

Makes Skeleton a unique node with the name specified in the `skeleton_name`. This allows the animation track paths to be unified independent of the scene hierarchy.

Rest Fixer

Reference poses defined in SkeletonProfileHumanoid have the following rules:

- The humanoid is T-pose

- The humanoid is facing +Z in the Right-Handed Y-UP Coordinate System
- The humanoid should not have a Transform as Node
- Directs the +Y axis from the parent joint to the child joint
- +X rotation bends the joint like a muscle contracting

These rules are convenient definitions for blend animation and Inverse Kinematics (IK). If your model does not match this definition, you need to fix it with these options.

Apply Node Transform

If the asset is not exported correctly for sharing, the imported Skeleton may have a Transform as a Node. For example, a glTF exported from Blender with no "Apply Transform" executed is one such case. It looks like the model matches the definition, but the internal Transforms are different from the definition. This option fixes such models by applying Transforms on import.

Note

If the imported scene contains objects other than Skeletons, this option may have a negative effect.

Normalize Position Tracks

Position track is used mostly for model movement, but sharing the moving animation between models with different heights may cause the appearance of slipping due to the difference in stride length. This option normalizes the Position track values based on the `scale_base_bone` height. The `scale_base_bone` height is stored in the Skeleton as the `motion_scale`, and the normalized Position track values is multiplied by that value on playback. If this option is disabled, the Position tracks is not normalized and the Skeleton's `motion_scale` is always imported as `1.0`.

With SkeletonProfileHumanoid, `scale_base_bone` is "Hips", therefore the Hips' height is used as the `motion_scale`.

Overwrite Axis

Unifies the models' Bone Rests by overwriting it to match the reference poses defined in the SkeletonProfile.

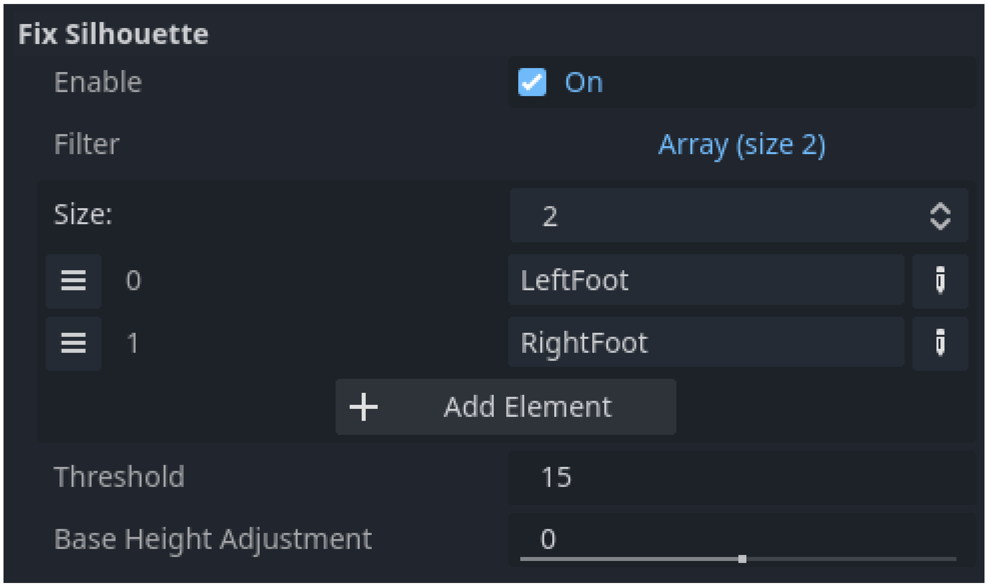
Note

This is the most important option for sharing animations in Godot 4.0+, but be aware that this option can produce horrible results **if the original Bone Rest set externally is important**. If you want to share animations with keeping the original Bone Rest, consider to use the Realtime Retarget Module [https://github.com/TokageItLab/realtime_retarget].

Fix Silhouette

Attempts to make the model's silhouette match that of the reference poses defined in the SkeletonProfile, such as T-Pose. This cannot fix silhouettes which are too different, and it may not work for fixing bone roll.

With SkeletonProfileHumanoid, this option does not need to be enabled for T-pose models, but should be enabled for A-pose models. However in that case, the fixed foot results may be bad depending on the heel height of the model, so it may be necessary to add the SkeletonProfile bone names you do not want fixed in the `filter` array, as in the below example.



Fix Silhouette

Enable ☑ On

Filter Array (size 2)

Size: 2

0 LeftFoot

1 RightFoot

Threshold 15

Base Height Adjustment 0

Also, for models with bent knees or feet, it may be necessary to adjust the `scale_base_bone` height. For that, you can use `base_height_adjustment` option.

Exporting 3D scenes

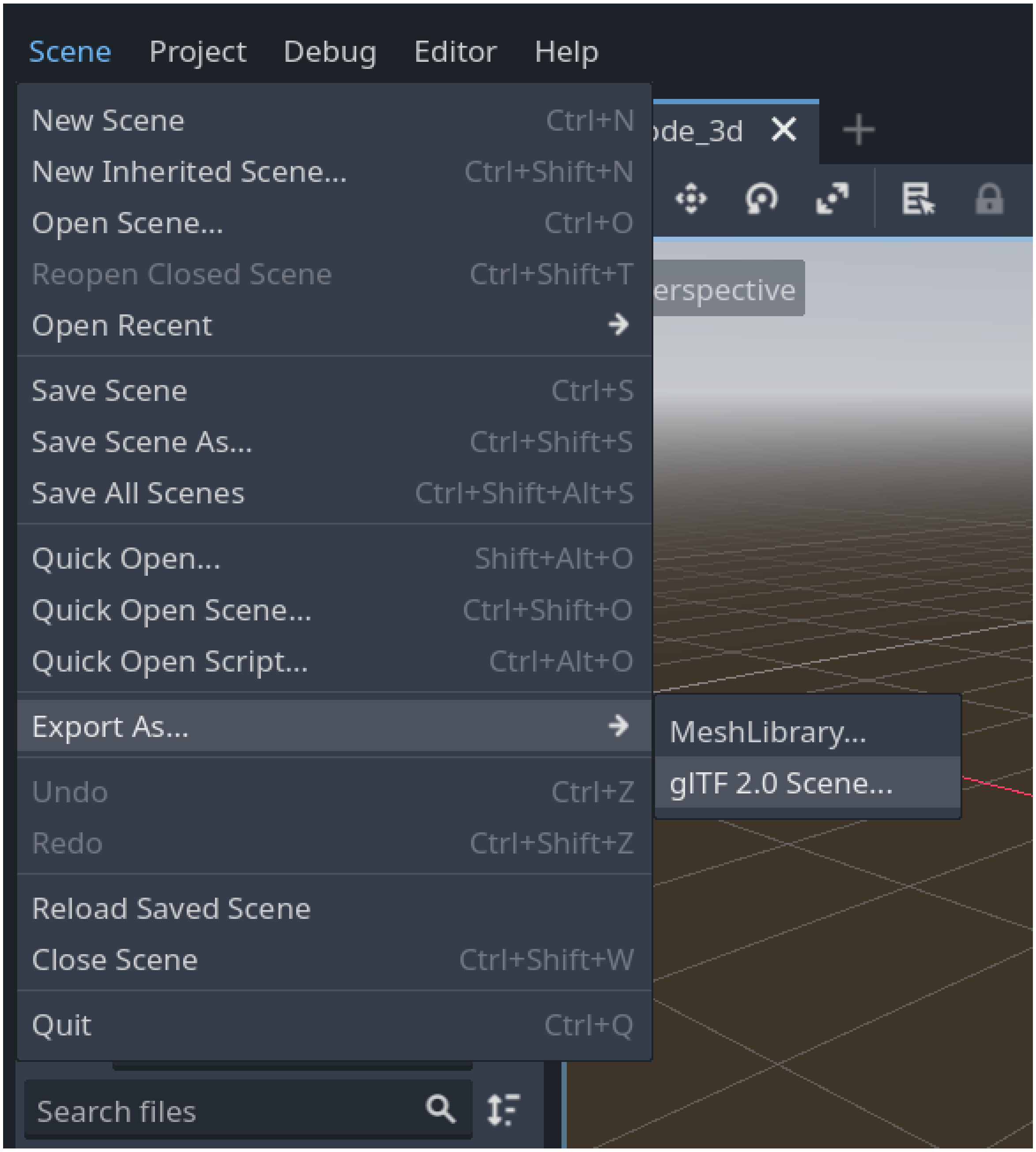
Overview

In Godot, it is possible to export 3D scenes as a glTF 2.0 file. You can export as a glTF binary (`.glb` file) or glTF embedded with textures (`gltf` + `.bin` + textures). This allows you to create scenes in Godot, such as a CSG mesh blockout for a level, export it to clean it up in a program such as Blender, and then bring it back into Godot.

Note

Only Blender 2.83 and newer can import glTF files exported by Godot.

To export a scene in the editor go to **Scene > Export As... > glTF 2.0 Scene...**


Scene
Project
Debug
Editor
Help
New Scene Ctrl+N
New Inherited Scene... Ctrl+Shift+N
Open Scene... Ctrl+O
Reopen Closed Scene Ctrl+Shift+T
Open Recent
Save Scene Ctrl+S
Save Scene As... Ctrl+Shift+S
Save All Scenes Ctrl+Shift+Alt+S
Quick Open... Shift+Alt+O
Quick Open Scene... Ctrl+Shift+O
Quick Open Script... Ctrl+Alt+O
Export As...
MeshLibrary...
glTF 2.0 Scene...
Undo Ctrl+Z
Redo Ctrl+Shift+Z
Reload Saved Scene
Close Scene Ctrl+Shift+W
Quit Ctrl+Q
ode_3d
erspective
Search files

Limitations

There are several limitations with glTF export.

- No support for exporting particles since their implementation varies across engines.
- ShaderMaterials cannot be exported.

- No support for exporting 2D scenes.

See also

3D scenes can be saved at runtime using runtime file loading and saving, including from an exported project.

Blender ESCN exporter

Note

This chapter relates to the Blender plugin called "Godot Blender Exporter", which can be downloaded here: https://github.com/godotengine/godot-blender-exporter

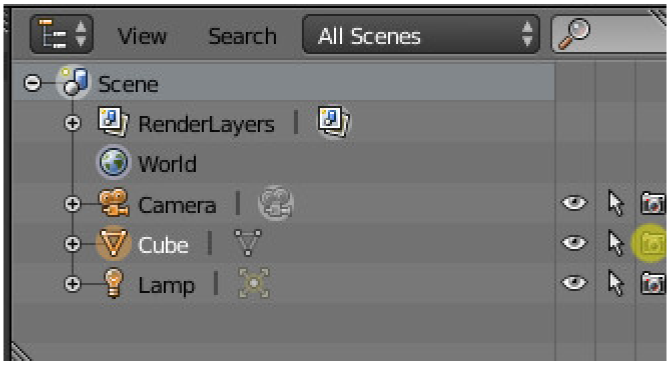
This plugin can be used to export Blender scenes in a Godot-specific scene format called ESCN, which is similar to TSCN (text format) but will be imported as binary SCN for performance.

Details on exporting

- Materials
- Physics properties
- Lights
- Mesh
- Skeleton
- Animation

Disabling specific objects

Sometimes you don't want some objects exported (e.g. high-res models used for baking). An object will not be exported if it is not rendered in the scene. This can be set in the outliner:


View
Search
All Scenes
Scene
RenderLayers
World
Camera
Cube
Lamp

Objects hidden in the viewport will be exported, but will be hidden in the Godot scene.

Build pipeline integration

If you have hundreds of model files, you don't want your artists to waste time manually exporting their blend files. To combat this, the exporter provides a Python function `io_scene_godot.export(out_file_path)` that can be called to export a file. This allows easy integration with other build systems. An example Makefile and Python script that exports all the blends in a directory are present in the godot-blender-exporter repository [https://github.com/godotengine/godot-blender-exporter].

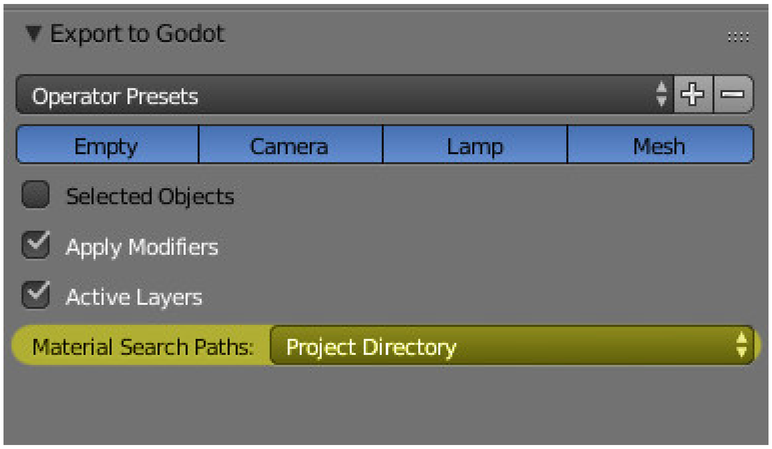
Materials

Using existing Godot materials

One way in which the exporter can handle materials is to attempt to match the Blender material with an existing Godot material. This has the advantage of being able to use all of the features of Godot's material system, but it means that you cannot see your model with the material applied inside Blender.

To do this, the exporter attempts to find Godot materials with names that match those of the material name in Blender. So if you export an object in Blender with the material name `PurpleDots` then the exporter will search for the file `PurpleDots.tres` and assign it to the object. If this file is not a `StandardMaterial3D` or `ShaderMaterial` or if it cannot be found, then the exporter will fall back to exporting the material from Blender.

Where the exporter searches for the `.tres` file is determined by the "Material Search Paths" option:



☐ Selected Objects
☑ Apply Modifiers
☑ Active Layers
Material Search Paths: Project Directory

This can take the value of:

- Project Directory - Attempts to find the `project.Godot` and recursively searches through subdirectories. If `project.Godot` cannot be found it will throw an error. This is useful for most projects where naming conflicts are unlikely.
- Export Directory - Look for materials in subdirectories of the export location. This is useful for projects where you may have duplicate material names and need more control over what material gets assigned.
- None - Do not search for materials. Export them from the Blender file.

Export of Cycles/EEVEE materials

The exporter has a primitive support for converting Cycles/EEVEE material node tree to Godot Shader Material. Note that some of the Shader Node are not supported yet due to difficulties in implementation, which are:

- all the `noisy textures`
- `generated texture coordinates`
- `group node`
- shader nodes except `PrincipledBSDF`, `Diffuse`, `Glossy`, `Glass`, `add shader` and `mix shader`

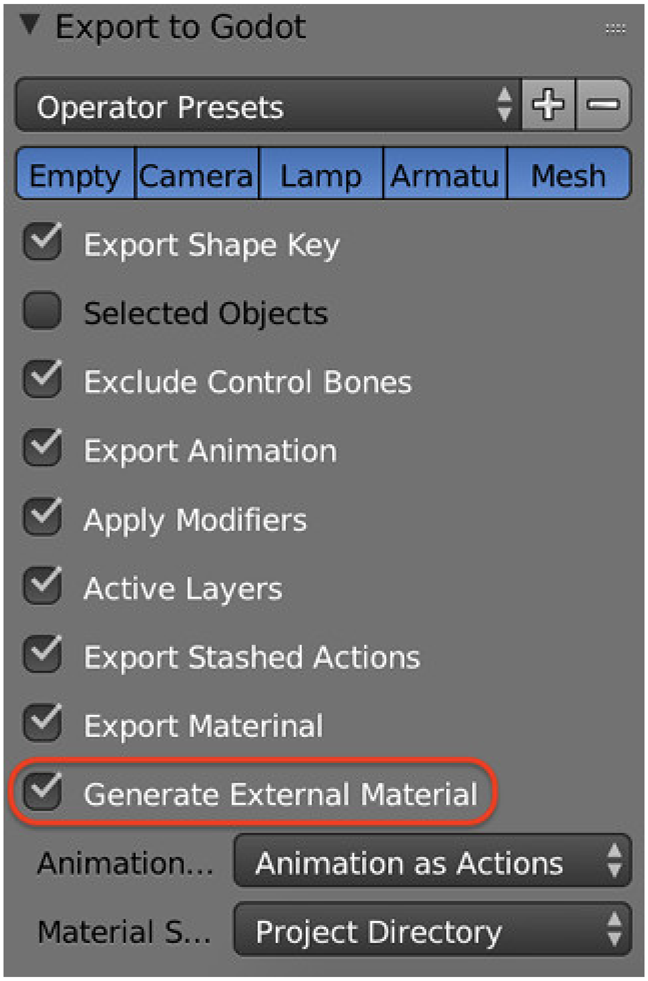
Warning

If possible, try to use PrincipledBSDF node with GGX distribution as the output shader node, it is the only one guaranteed to be exactly correct. Others are just based on approximation.

Sometimes materials may not be valid for exporting (e.g. has some unsupported node) or it is using Blender Internal Engine, only the diffuse color and a few flags (e.g. unshaded) are exported and form a StandardMaterial3D.

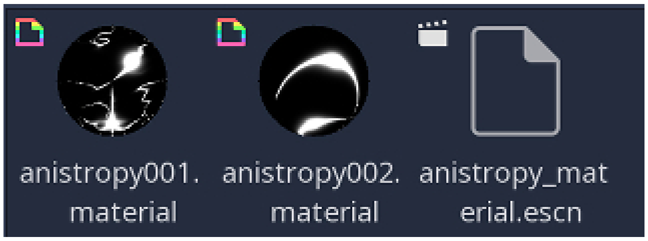
Generate external materials

The default configuration of material exporting would keep all the materials internal to the `escn` file. There is an option which could enable generating external `.material` file when the `escn` file opens in Godot.



☑ Export Shape Key
☐ Selected Objects
☑ Exclude Control Bones
☑ Export Animation
☑ Apply Modifiers
☑ Active Layers
☑ Export Stashed Actions
☑ Export Materinal
☑ Generate External Material
Animation... Animation as Actions
Material S... Project Directory

`.material` file can be assigned to any material slot to be an external resource.



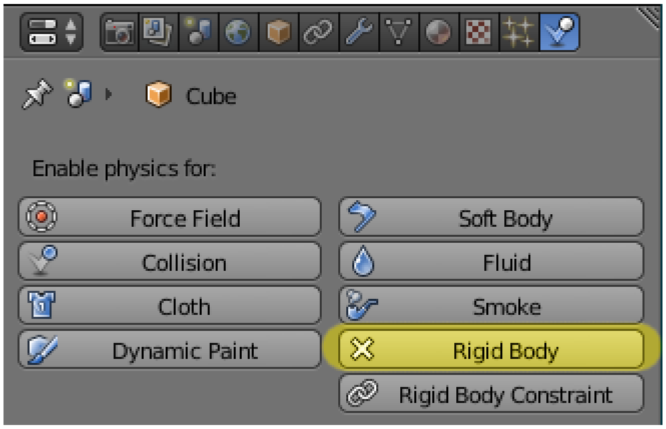
anistropy001.material

anistropy002.material

anistropy_material.escn

Physics properties

Exporting physics properties is done by enabling "Rigid Body" in Blender's physics tab:

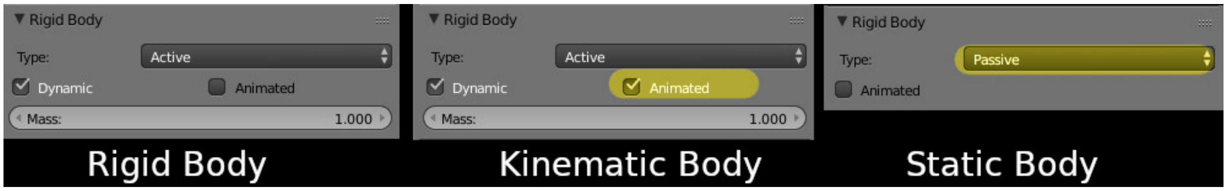

Cube
Enable physics for:
Force Field
Soft Body
Collision
Fluid
Cloth
Smoke
Dynamic Paint
Rigid Body
Rigid Body Constraint

Important

By default, a single Blender object with rigid body enabled will export as three nodes: a PhysicsBody, a CollisionShape, and a MeshInstance3D.

Body type

Blender only has the concept of "Active" and "Passive" rigid bodies. These turn into Static and RigidBody nodes. To create a kinematic body, enable the "animated" checkbox on an "Active" body:



▼ Rigid Body
Type: Active
☑ Dynamic ☐ Animated
Mass: 1.000

Rigid Body

▼ Rigid Body
Type: Active
☑ Dynamic ☑ Animated
Mass: 1.000

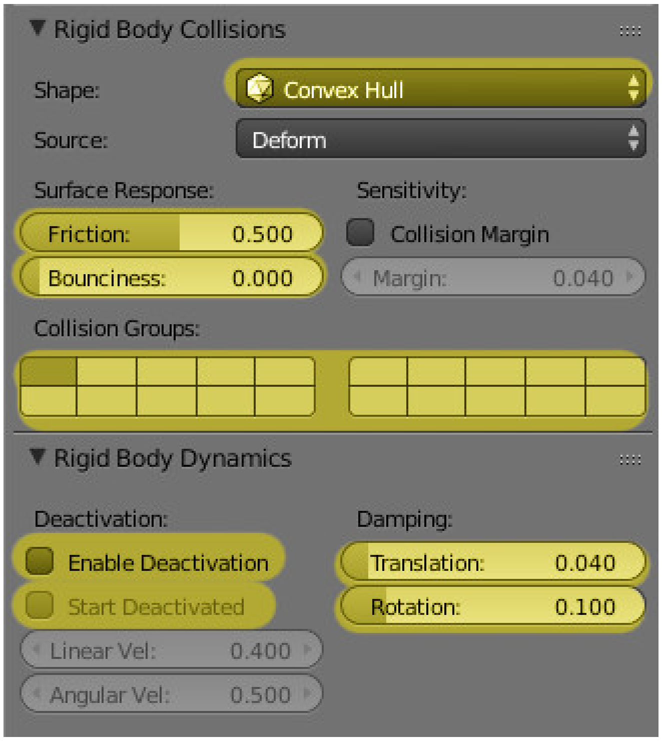
Kinematic Body

▼ Rigid Body
Type: Passive
☐ Animated

Static Body

Collision shapes

Many of the parameters for collision shapes are missing from Blender, and many of the collision shapes are also not present. However, almost all of the options in Blender's rigid body collision and rigid body dynamics interfaces are supported:



▼ Rigid Body Collisions

Shape: Convex Hull

Source: Deform

Surface Response:

Friction: 0.500

Bounciness: 0.000

Sensitivity:

☐ Collision Margin

Margin: 0.040

Collision Groups:

▼ Rigid Body Dynamics

Deactivation:

☐ Enable Deactivation

☐ Start Deactivated

Linear Vel: 0.400

Angular Vel: 0.500

Damping:

Translation: 0.040

Rotation: 0.100

There are the following caveats:

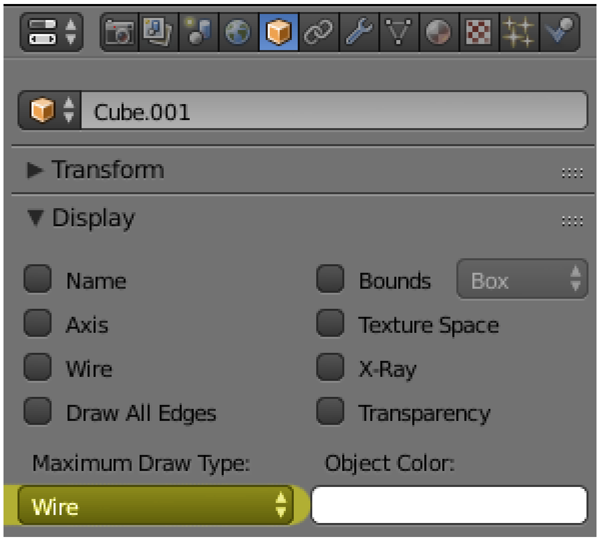
- Not all of the collision shapes are supported. Only `Mesh`, `Convex Hull`, `Capsule`, `Sphere` and `Box` are supported in both Blender and Godot
- In Godot, you can have different collision groups and collision masks. In Blender you only have collision groups. As a result, the exported object's collision mask is equal to its collision group. Most of the time, this is what you want.

Important

To build compound physics shapes, parent together multiple objects with rigid body enabled. The physics properties are taken from the parent-most rigid body, and the rest are used as collision shapes.

Collision geometry only

Frequently you want different geometry for your collision meshes and your graphical meshes, but by default, the exporter will export a mesh along with the collision shape. To only export the collision shape, set the object's maximum draw type to Wire:



Cube.001

☐ Name ☐ Bounds: Box
☐ Axis ☐ Texture Space
☐ Wire ☐ X-Ray
☐ Draw All Edges ☐ Transparency

Maximum Draw Type: Wire

This will also influence how the object is shown in Blender's viewport. Most of the time, you want your collision geometry to be shown see-through when working on the models, so this works out fairly nicely.

Lights

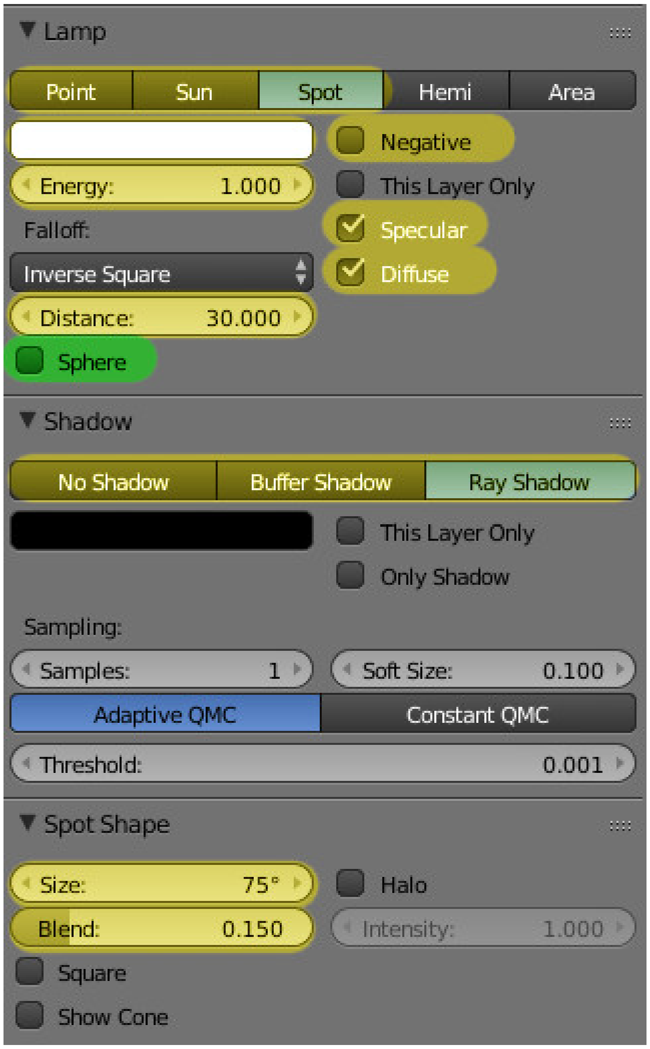
Warning

By default, lamps in Blender have shadows enabled. This can cause performance issues in Godot.

Warning

Lamps are exported using their "Blender Render" settings. When Blender 2.8 is released, this will be removed and this part of the exporter will change.

Sun, point and spot lamps are all exported from Blender along with many of their properties:



Lamp
Point
Sun
Spot
Hemi
Area

☐ Negative

Energy: 1.000

☐ This Layer Only

☑ Specular

☑ Diffuse

Falloff: Inverse Square

Distance: 30.000

☐ Sphere

Shadow
No Shadow
Buffer Shadow
Ray Shadow

☐ This Layer Only

☐ Only Shadow

Sampling:

Samples: 1

Soft Size: 0.100

Adaptive QMC
Constant QMC

Threshold: 0.001

Spot Shape

Size: 75°

☐ Halo

Blend: 0.150

Intensity: 1.000

☐ Square

☐ Show Cone

There are some things to note:

- In Blender, a light casts light all the way to infinity. In Godot, it is clamped by the attenuation distance. To most closely match between the viewport and Godot, enable the "Sphere" checkbox. (Highlighted green)
- Light attenuation models differ between Godot and Blender. The exporter attempts to make them match, but it isn't always very good.
- Spotlight angular attenuation models also differ between Godot and Blender. The exporter attempts to make them similar, but it doesn't always look the same.

- There is no difference between buffer shadow and ray shadow in the export.

Mesh

Modifiers

There is an exporting option `Apply Modifiers` to control whether mesh modifiers are applied to the exported mesh.

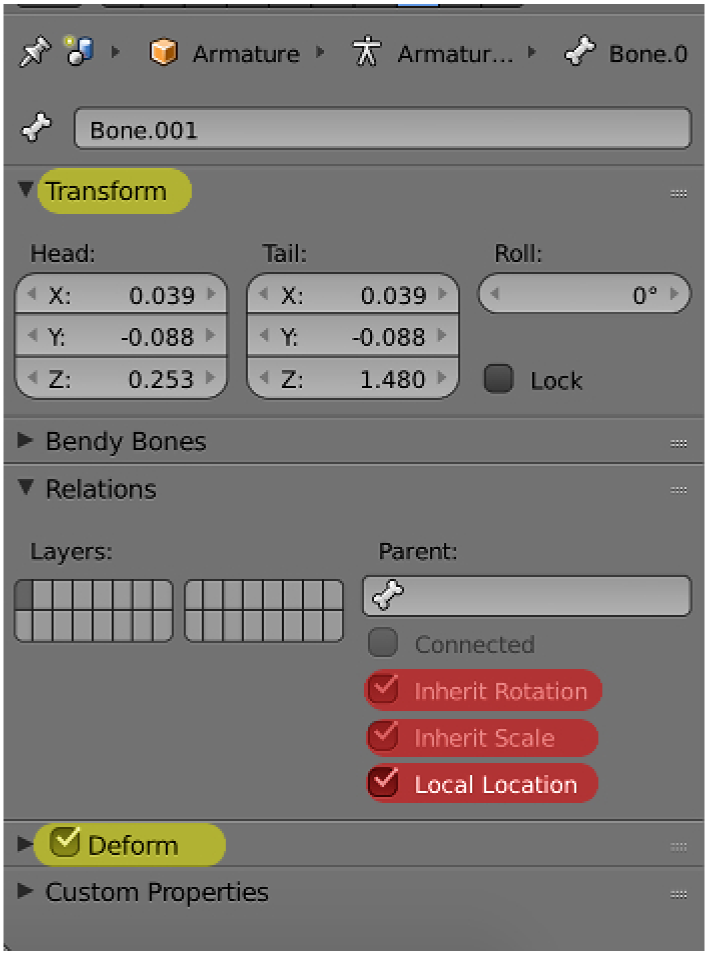
Shape Key

Exporting mesh shape key is supported, however exporting each shape key is almost like exporting the mesh again, so don't be surprised it takes a relatively long time.

Warning

A lot of modifiers are not compatible with shape keys (e.g. subsurface modifier), so if you found you have incorrect shape keys exported, try to disable `Apply Modifiers` and do the exporting again. Besides, it is worthwhile to report the incompatible modifier to the issue list [https://github.com/godotengine/godot-blender-exporter/issues], which helps to develop the exporter to have a more precise check of modifiers.

Skeleton



Armature
Armatur...
Bone.0
Transform
Bendy Bones
Relations
Deform
Custom Properties

Bone.001

Head: X: 0.039 Y: -0.088 Z: 0.253

Tail: X: 0.039 Y: -0.088 Z: 1.480

Roll: 0°

☐ Lock

Layers:

Parent:

☐ Connected

☑ Inherit Rotation

☑ Inherit Scale

☑ Local Location

☑ Deform

Rest Bone

Armature object in Blender is exported as a Skeleton node along with rest position (transform in Godot) of bones.

Warning

The three check boxes `Inherit Rotation`, `Inherit Scale`, `Local Location` (colored in red) must be ticked when building armature in

Blender, so that the exported bone transform be consistent between Blender and Godot

It is important that the mesh is not deformed by bones when exporting in Blender. Make sure that the skeleton is reset to its T-pose or default rest pose.

Bone Weights

Blender puts rigged mesh vertices which have no bone weights at their original position, but these vertices would be placed at (0, 0, 0) in Godot, making the mesh deformed. Therefore, the exporter will raise an error for any vertex with no bone weights detected in a rigged mesh.

Non-Deform Bone

Note that the non-deform bone can be configured as not exported by enabling the `Exclude Control Bones`; the deform bone checkbox is shown in the picture.

Bone Attachment

A bone can be the parent of an object in Blender; this relation is exported as a BoneAttachment node in the Godot scene.

Animation

Animation supported:

- transform animation of all types of objects
- transform animation of pose bones
- shape key animation
- light animation
- camera animation

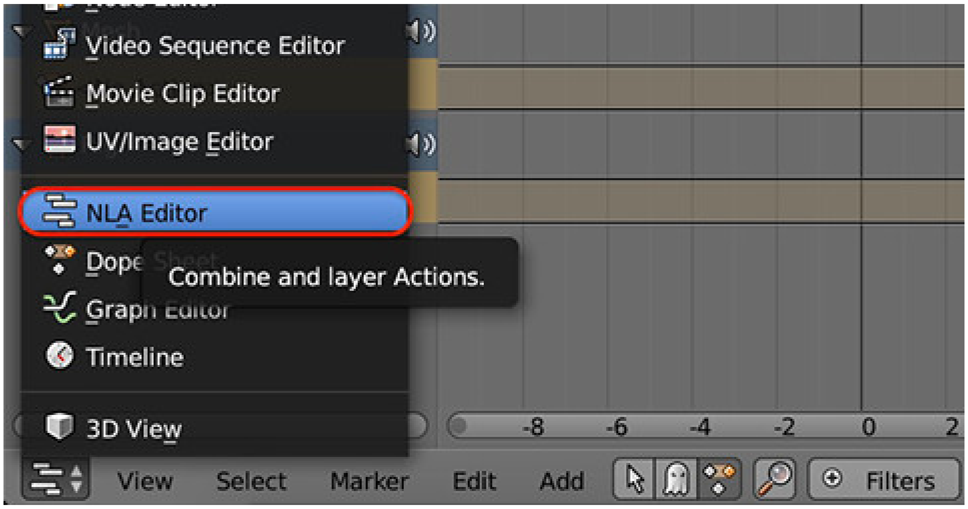
Multiple Actions For Single Object

In most games, one object would have several animations to switch between. This add-on has support for exporting multiple actions all at once into a single AnimationPlayer and makes it easy to switch actions.

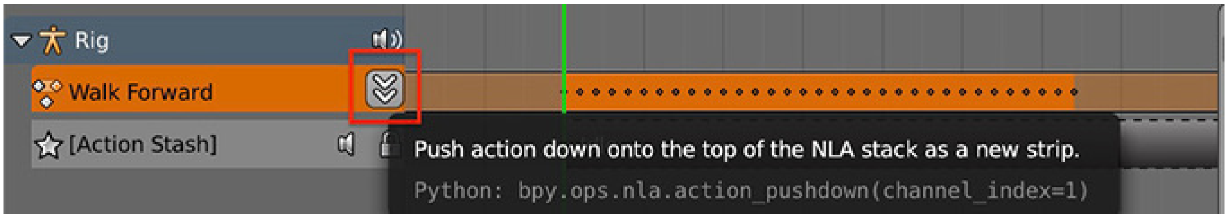
This workflow makes use of blender nla_tracks. Here is a brief guide on how to use this feature:

1. Stash active action

New created action is always an active action bound to object. There are several ways to place an active action into NLA track, one is of course doing it in `NLA Editor`


Video Sequence Editor
Movie Clip Editor
UV/Image Editor
NLA Editor
Dope Sheet
Graph Editor
Timeline
3D View
-8
-6
-4
-2
0
2
View
Select
Marker
Edit
Add
Filters

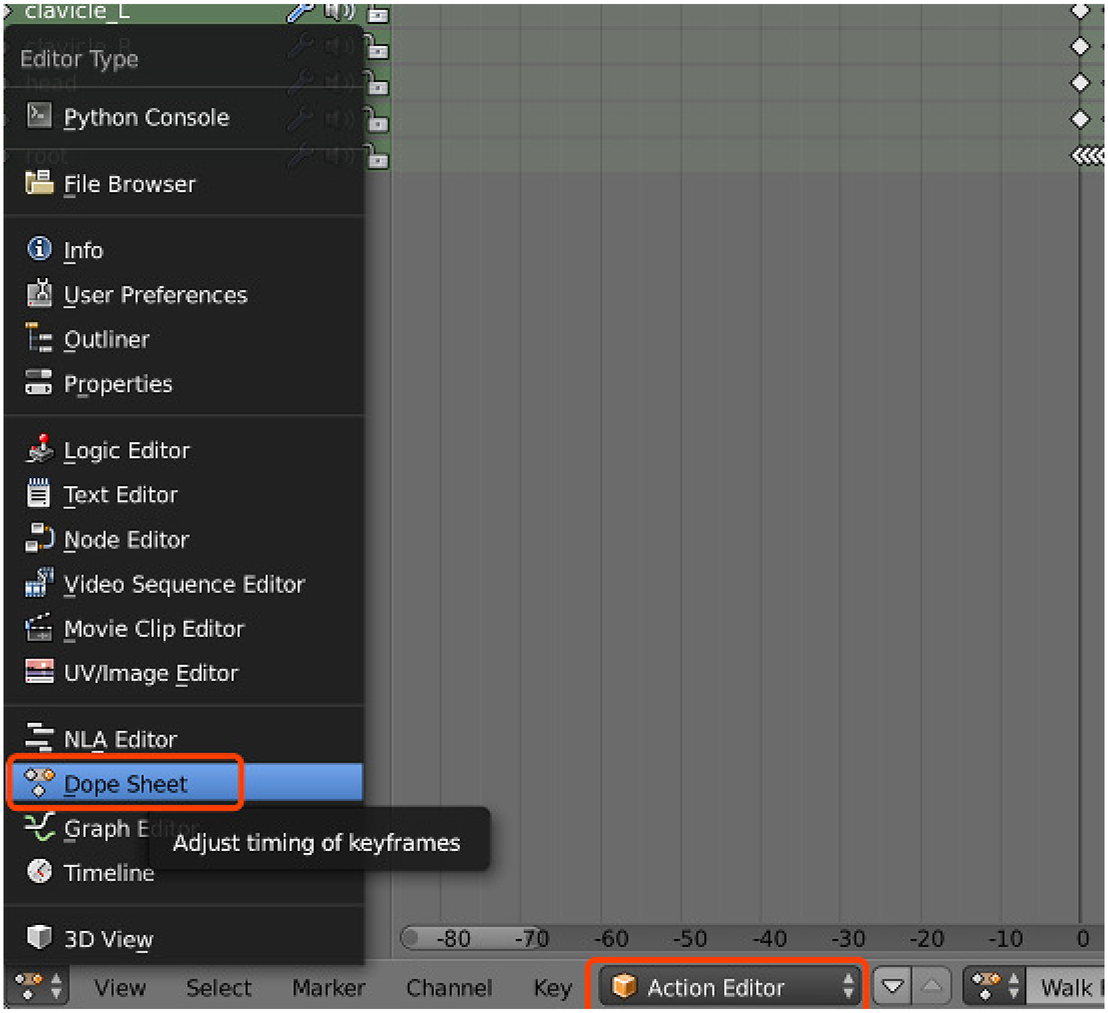
Combine and layer Actions.

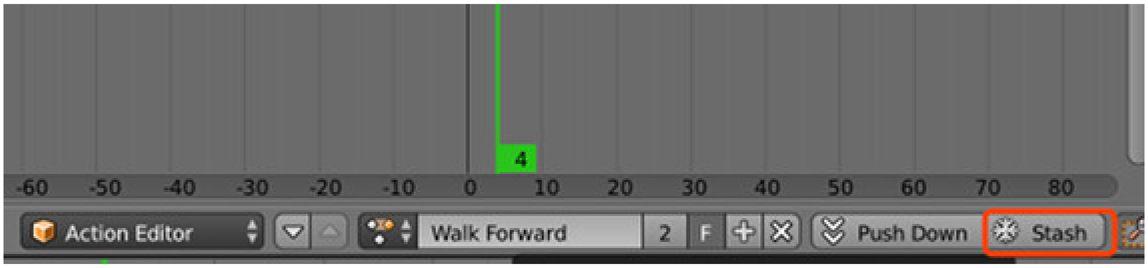

Rig
Walk Forward
[Action Stash]

Push action down onto the top of the NLA stack as a new strip.

```
Python: bpy.ops.nla.action_pushdown(channel_index=1)
```

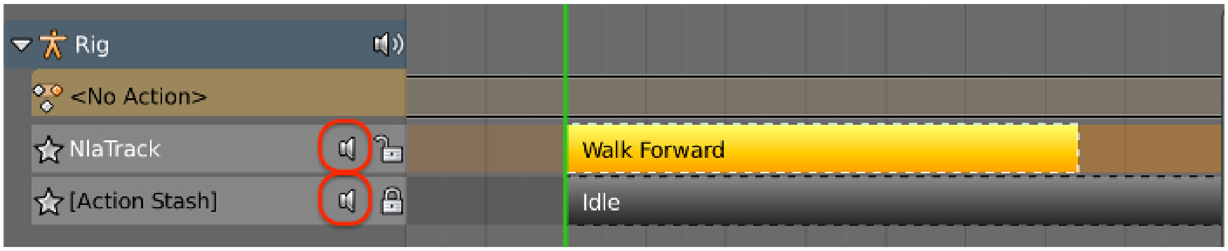
Or it can be done stashing the action in `Dope Sheet`


clavicle_L
Editor Type
Python Console
File Browser
Info
User Preferences
Outliner
Properties
Logic Editor
Text Editor
Node Editor
Video Sequence Editor
Movie Clip Editor
UV/Image Editor
NLA Editor
Dope Sheet
Graph Editor
Timeline
3D View
Adjust timing of keyframes
-80 -70 -60 -50 -40 -30 -20 -10 0
View
Select
Marker
Channel
Key
Action Editor
Walk


4
-60 -50 -40 -30 -20 -10 0 10 20 30 40 50 60 70 80
Action Editor
Walk Forward
2
F
Push Down
Stash

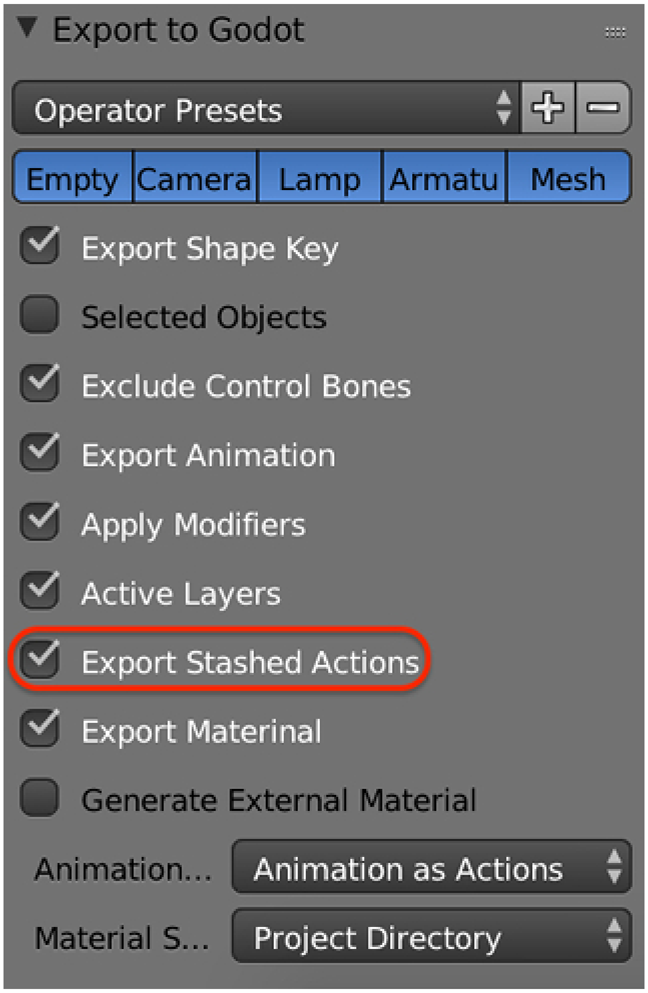
2. Check mute status of NLA tracks

An NLA track can be `mute` or `unmute`, the exporter will export all the `mute` NLA track as a separate action, while blends all the `unmute` NLA tracks into every action (including the action) being exported.


Rig
<No Action>
NlaTrack
[Action Stash]
Walk Forward
Idle

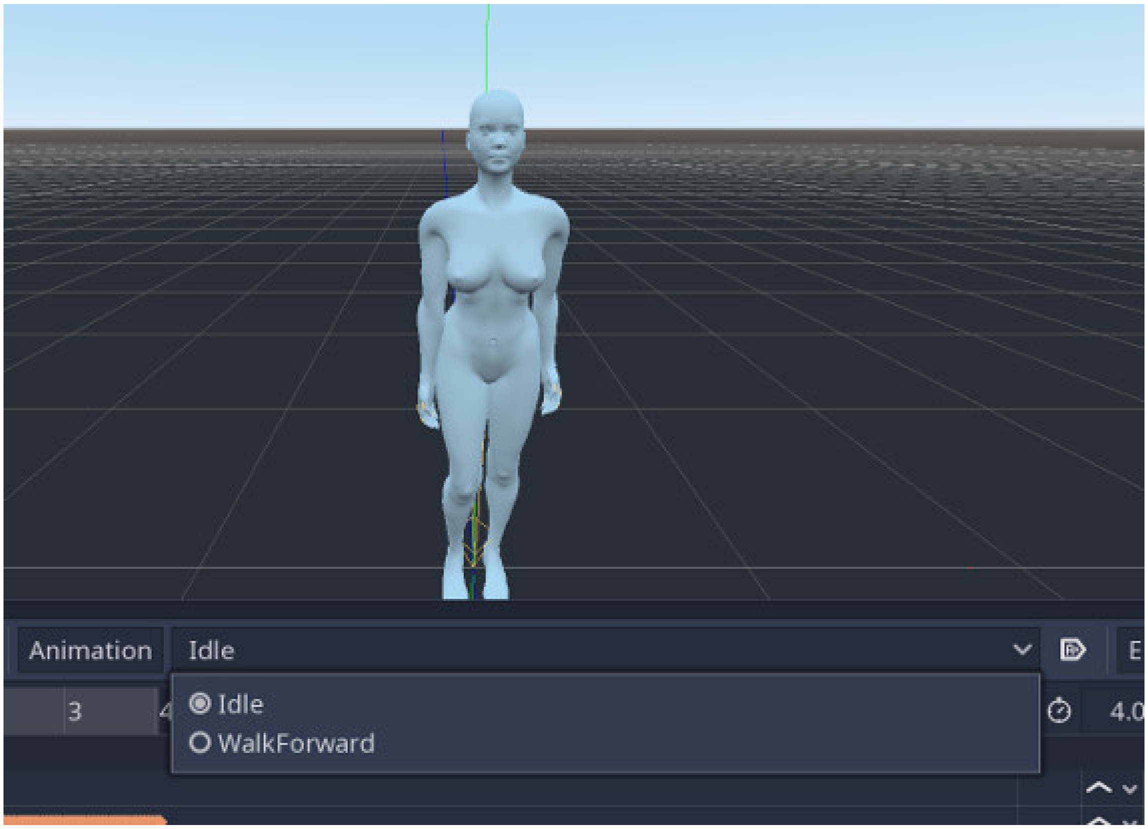
3. Export the scene

Make sure the `Export Stashed Actions` option has been turned on.


Export to Godot
Operator Presets
Empty
Camera
Lamp
Armatu
Mesh

☑ Export Shape Key
☐ Selected Objects
☑ Exclude Control Bones
☑ Export Animation
☑ Apply Modifiers
☑ Active Layers
☑ Export Stashed Actions
☑ Export Materinal
☐ Generate External Material
Animation... Animation as Actions
Material S... Project Directory

Then all the stashed actions, as well as the active action, are exported to an AnimationPlayer.



Constraints

Sometimes complicated animation is built with object constraint; a usual example is inverse kinematics. The add-on would automatically check if an object has some constraint; if it does, all the constraints are baked into actions and then exported along with the object.

Animation Mode

Godot and Blender have different structure to store animation data. In Godot animation data is stored in an AnimationPlayer node, instead of in each animated node. In order to fix this inconsistency and still make the animation play versatile, this add-on has three animation exporting modes.

Mode 'Animation as Actions'

Treat all the animations as object actions, so in the exported scene, every object would have its own AnimationPlayer and hold its actions.

Mode 'Scene Animation'

If you want your animation to generate the same result as playing at Blender's timeline, this is what you want. In this mode, all the animations in the scene are placed in just one AnimationPlayer in the scene root.

Mode 'Animation as Action with Squash'

This mode has very similar behavior of mode 'Animation as Action', but it can generate fewer AnimationPlayers; objects in parent-children relationship would share their AnimationPlayer. It is useful when you have several rigs, and each Skeleton and Mesh has actions; then one rig would have just one AnimationPlayer.

Audio

- Audio buses
- Audio effects
- Audio streams
- Sync the gameplay with audio and music
- Recording with microphone
- Text to speech

Audio buses

Introduction

Godot's audio processing code has been written with games in mind, with the aim of achieving an optimal balance between performance and sound quality.

Godot's audio engine allows any number of audio buses to be created and any number of effect processors can be added to each bus. Only the hardware of the device running your game will limit the number of buses and effects that can be used before performance starts to suffer.

Decibel scale

Godot's sound interface is designed to meet the expectations of sound design professionals. To this end, it primarily uses the decibel scale.

For those unfamiliar with it, it can be explained with a few facts:

- The decibel (dB) scale is a relative scale. It represents the ratio of sound power by using 20 times the base 10 logarithm of the ratio ($20 \times \log_{10}(P/P_0)$).
- For every 6 dB, sound amplitude doubles or halves. 12 dB represents a factor of 4, 18 dB a factor of 8, 20 dB a factor of 10, 40 dB a factor of 100, etc.
- Since the scale is logarithmic, true zero (no audio) can't be represented.
- 0 dB is the maximum amplitude possible in a digital audio system. This limit is not the human limit, but a limit from the

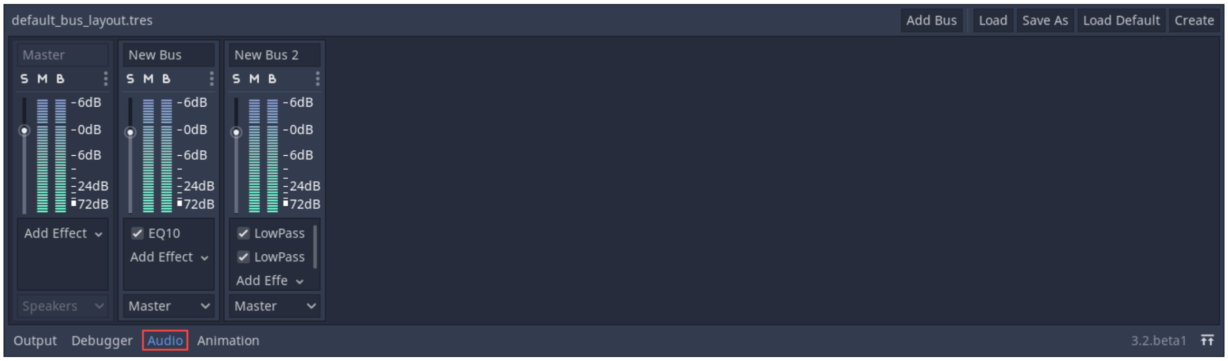
sound hardware. Audio with amplitudes that are too high to be represented properly below 0 dB create a kind of distortion called *clipping*.

- To avoid clipping, your sound mix should be arranged so that the output of the *master bus* (more on that later) never exceeds 0 dB.
- Every 6 dB below the 0 dB limit, sound energy is *halved*. It means the sound volume at -6 dB is half as loud as 0dB. -12 dB is half as loud as -6 dB and so on.
- When working with decibels, sound is considered no longer audible between -60 dB and -80 dB. This makes your working range generally between -60 dB and 0 dB.

This can take a bit getting used to, but it's friendlier in the end and will allow you to communicate better with audio professionals.

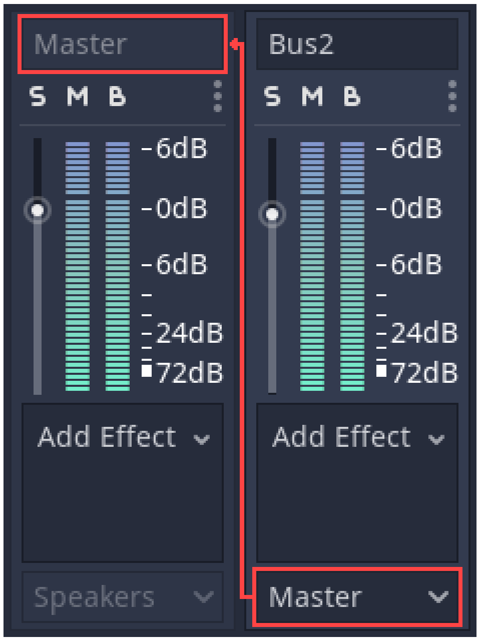
Audio buses

Audio buses can be found in the bottom panel of the Godot editor:


default_bus_layout.tres
Add Bus
Load
Save As
Load Default
Create
Master
New Bus
New Bus 2
S M B
-6dB
-0dB
-6dB
24dB
72dB
Add Effect
EQ10
LowPass
LowPass
Add Effe
Speakers
Master
Master
Output
Debugger
Audio
Animation
3.2.beta1

An *audio bus* (also called an *audio channel*) can be considered a place that audio is channeled through on the way to playback through a device's speakers. Audio data can be *modified* and *re-routed* by an audio bus. An audio bus has a VU meter (the bars that light up when sound is played) which indicates the amplitude of the signal passing through.

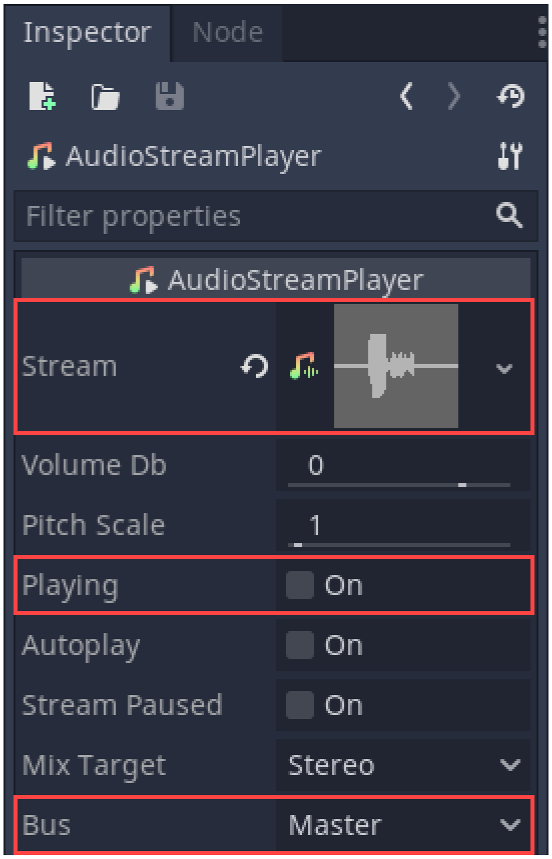
The leftmost bus is the *master bus*. This bus outputs the mix to your speakers so, as mentioned in the *Decibel scale* section above, make sure that your mix level doesn't reach 0 dB in this bus. The rest of the audio buses can be flexibly routed. After modifying the sound, they send it to another bus to the left. The destination bus can be specified for each of the non-master audio buses. Routing always passes audio from buses on the right to buses further to the left. This avoids infinite routing loops.


Master
S M B
-6dB
-0dB
-6dB
-24dB
-72dB
Add Effect
Speakers
Bus2
S M B
-6dB
-0dB
-6dB
-24dB
-72dB
Add Effect
Master

In the above image, the output of *Bus 2* has been routed to the *Master* bus.

Playback of audio through a bus

To test passing audio to a bus, create an AudioStreamPlayer node, load an AudioStream and select a target bus for playback:


Inspector
Node
AudioStreamPlayer
Filter properties
AudioStreamPlayer

Stream
Volume Db 0
Pitch Scale 1
Playing ☐ On
Autoplay ☐ On
Stream Paused ☐ On
Mix Target Stereo
Bus Master

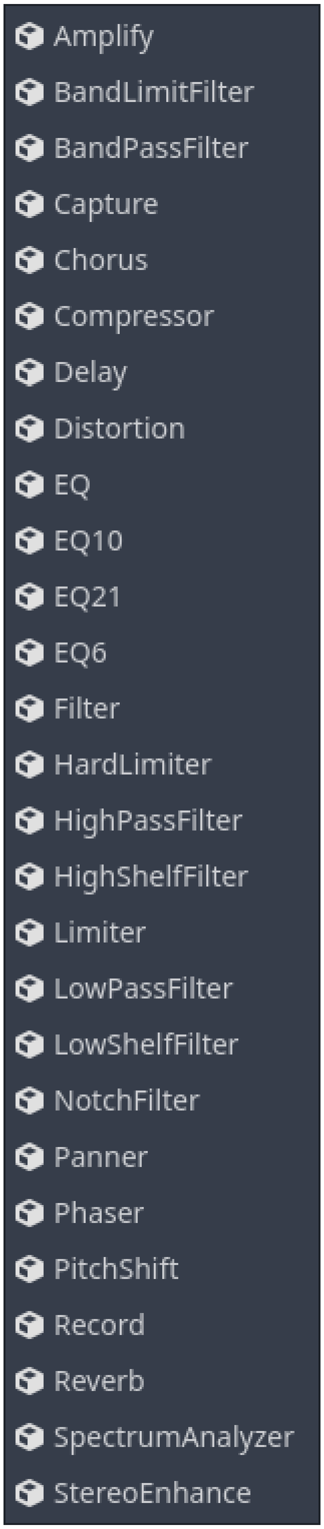
Finally, toggle the **Playing** property to **On** and sound will flow.

See also

You may also be interested in reading about Audio streams now.

Adding effects

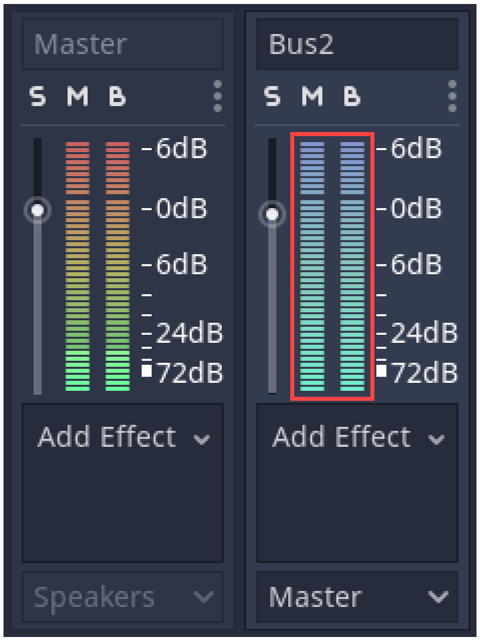
Audio buses can contain all sorts of effects. These effects modify the sound in one way or another and are applied in order.


Amplify
BandLimitFilter
BandPassFilter
Capture
Chorus
Compressor
Delay
Distortion
EQ
EQ10
EQ21
EQ6
Filter
HardLimiter
HighPassFilter
HighShelfFilter
Limiter
LowPassFilter
LowShelfFilter
NotchFilter
Panner
Phaser
PitchShift
Record
Reverb
SpectrumAnalyzer
StereoEnhance

For information on what each effect does, see Audio effects.

Automatic bus disabling

There is no need to disable buses manually when not in use. Godot detects that the bus has been silent for a few seconds and disables it (including all effects).


Master
S M B
-6dB
-0dB
-6dB
-24dB
-72dB
Add Effect
Speakers
Bus2
S M B
-6dB
-0dB
-6dB
-24dB
-72dB
Add Effect
Master

Disabled buses have a blue VU meter instead of a red-green one.

Bus rearrangement

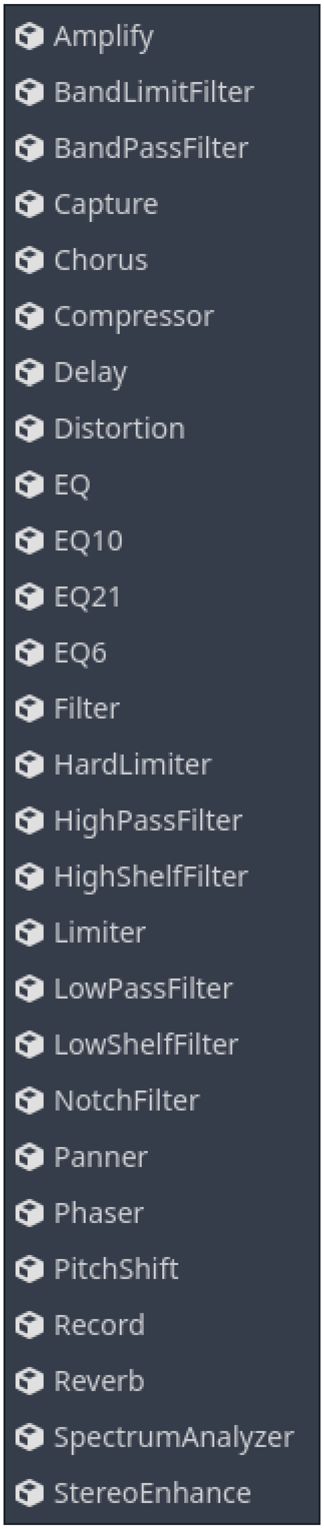
Stream Players use bus names to identify a bus, which allows adding, removing and moving buses around while the reference to them is kept. However, if a bus is renamed, the reference will be lost and the Stream Player will output to Master. This system was chosen because rearranging buses is a more common process than renaming them.

Default bus layout

The default bus layout is automatically saved to the `res://default_bus_layout.tres` file. Custom bus arrangements can be saved and loaded from disk.

Audio effects

Godot includes several audio effects that can be added to an audio bus to alter every sound file that goes through that bus.



Amplify
BandLimitFilter
BandPassFilter
Capture
Chorus
Compressor
Delay
Distortion
EQ
EQ10
EQ21
EQ6
Filter
HardLimiter
HighPassFilter
HighShelfFilter
Limiter
LowPassFilter
LowShelfFilter
NotchFilter
Panner
Phaser
PitchShift
Record
Reverb
SpectrumAnalyzer
StereoEnhance

Try them all out to get a sense of how they alter sound. Here follows a short description of the available effects:

Amplify

Amplify changes the volume of the signal. Some care needs to be taken, though: setting the level too high can make the sound digitally clip, which can produce unpleasant crackles and pops.

BandLimit and BandPass

These are resonant filters which block frequencies around the *Cutoff* point. BandPass can be used to simulate sound passing through an old telephone line or megaphone. Modulating the BandPass frequency can simulate the sound of a wah-wah guitar pedal, think of the guitar in Jimi Hendrix's *Voodoo Child (Slight Return)*.

Capture

The Capture effect copies the audio frames of the audio bus that it is on into an internal buffer. This can be used to capture data from the microphone or to transmit audio over the network in real-time.

Chorus

As the name of the effect implies, the Chorus effect makes a single audio sample sound like an entire chorus. It does this by duplicating a signal and very slightly altering the timing and pitch of each duplicate, and varying that over time via an LFO (low frequency oscillator). The duplicate(s) are then mixed back together with the original signal, producing a lush, wide, and large sound. Although chorus is traditionally used for voices, it can be desirable with almost any type of sound.

Compressor

A dynamic range compressor automatically attenuates (ducks) the level of the incoming signal when its amplitude exceeds a certain threshold. The level of attenuation applied is proportional to how far the incoming audio exceeds the threshold. The compressor's Ratio parameter controls the degree of attenuation. One of the main uses of a compressor is to reduce the dynamic range of signals with very loud and quiet parts. Reducing the dynamic range of a signal can make it fit more comfortably in a mix.

The compressor has many uses. For example:

- It can be used in the Master bus to compress the whole output prior to being hit by a limiter, making the effect of the limiter much more subtle.
- It can be used in voice channels to ensure they sound as even as possible.
- It can be *sidechained* by another sound source. This means it can reduce the sound level of one signal using the level of another audio bus for threshold detection. This technique is very common in video game mixing to "duck" the level of music or sound effects when in-game or multiplayer voices need to be fully audible.
- It can accentuate transients by using a slower attack. This can make sound effects more punchy.

Note

If your goal is to prevent a signal from exceeding a given amplitude altogether, rather than to reduce the dynamic range of the signal, a limiter is likely a better choice than a compressor for this purpose. However, applying compression before a limiter is still good practice.

Delay

Digital delay essentially duplicates a signal and repeats it at a specified speed with a volume level that decays for each repeat. Delay is great for simulating the acoustic space of a canyon or large room, where sound bounces have a lot of *delay* between their repeats. This is in contrast to reverb, which has a more natural and blurred sound to it. Using this in conjunction with reverb can create very natural sounding environments!

Distortion

Makes the sound distorted. Godot offers several types of distortion:

- *Overdrive* sounds like a guitar distortion pedal or megaphone. Sounds distorted with this sound like they're coming through a low-quality speaker or device.
- *Tan* sounds like another interesting flavor of overdrive.
- *Bit crushing* clamps the amplitude of the signal, making it sound flat and crunchy.

All three types of distortion can add higher frequency sounds to an original sound, making it stand out better in a mix.

EQ

EQ is what all other equalizers inherit from. It can be extended with Custom scripts to create an equalizer with a custom number of bands.

EQ6, EQ10, EQ21

Godot provides three equalizers with different numbers of bands, which are represented in the title (6, 10, and 21 bands, respectively). An equalizer on the Master bus can be useful for cutting low and high frequencies that the device's speakers can't reproduce well. For example, phone or tablet speakers usually don't reproduce low frequency sounds well, and could make a limiter or compressor attenuate sounds that aren't even audible to the user anyway.

Note: The equalizer effect can be disabled when headphones are plugged in, giving the user the best of both worlds.

Filter

Filter is what all other filters inherit from and should not be used directly.

HardLimiter

A limiter is similar to a compressor, but it's less flexible and designed to prevent a signal's amplitude exceeding a given dB threshold. Adding a limiter to the final point of the Master bus is good practice, as it offers an easy safeguard against clipping.

HighPassFilter

Cuts frequencies below a specific *Cutoff* frequency. HighPassFilter is used to reduce the bass content of a signal.

HighShelfFilter

Reduces all frequencies above a specific *Cutoff* frequency.

Limiter

This is the old limiter effect, and it is recommended to use the new HardLimiter effect instead.

Here is an example of how this effect works, if the ceiling is set to -12 dB, and the threshold is 0 dB, all samples going through get reduced by 12dB. This changes the waveform of the sound and introduces distortion.

This effect is being kept to preserve compatibility, however it should be considered deprecated.

LowPassFilter

Cuts frequencies above a specific *Cutoff* frequency and can also resonate (boost frequencies close to the *Cutoff* frequency). Low pass filters can be used to simulate "muffled" sound. For instance, underwater sounds, sounds blocked by walls, or distant sounds.

LowShelfFilter

Reduces all frequencies below a specific *Cutoff* frequency.

NotchFilter

The opposite of the BandPassFilter, it removes a band of sound from the frequency spectrum at a given *Cutoff* frequency.

Panner

The Panner allows the stereo balance of a signal to be adjusted between the left and right channels. Headphones are recommended when configuring in this effect.

Phaser

This effect is formed by de-phasing two duplicates of the same sound so they cancel each other out in an interesting way. Phaser produces a pleasant whooshing sound that moves back and forth through the audio spectrum, and can be a great way to create sci-fi effects or Darth Vader-like voices.

PitchShift

This effect allows the adjustment of the signal's pitch independently of its speed. All frequencies can be increased/decreased with minimal effect on transients. PitchShift can be useful to create unusually high or deep voices. Do note that altering pitch can sound unnatural when pushed outside of a narrow window.

Record

The Record effect allows the user to record sound from a microphone.

Reverb

Reverb simulates rooms of different sizes. It has adjustable parameters that can be tweaked to obtain the sound of a specific room. Reverb is commonly outputted from Area3Ds (see Reverb buses), or to apply a "chamber" feel to all sounds.

SpectrumAnalyzer

This effect doesn't alter audio, instead, you add this effect to buses you want a spectrum analysis of. This would typically be used for audio visualization. Visualizing voices can be a great way to draw attention to them without just increasing their volume. A demo project using this can be found here [https://github.com/godotengine/godot-demo-projects/tree/master/audio/spectrum].

StereoEnhance

This effect uses a few algorithms to enhance a signal's stereo width.

Audio streams

Introduction

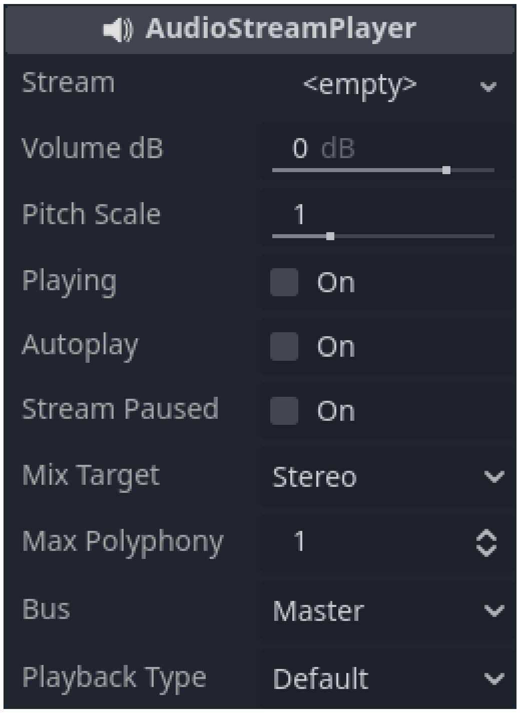
As you might have already read in Audio buses, sound is sent to each bus via an AudioStreamPlayer node. There are different kinds of AudioStreamPlayers. Each one loads an AudioStream and plays it back.

AudioStream

An audio stream is an abstract object that emits sound. The sound can come from many places, but is most commonly loaded from the filesystem. Audio files can be loaded as AudioStreams and placed inside an AudioStreamPlayer. You can find information on supported formats and differences in Importing audio samples.

There are other types of AudioStreams, such as AudioStreamRandomizer. This one picks a different audio stream from a list of streams each time it's played back, and applies random pitch and volume shifting. This can be helpful for adding variation to sounds that are played back often.

AudioStreamPlayer



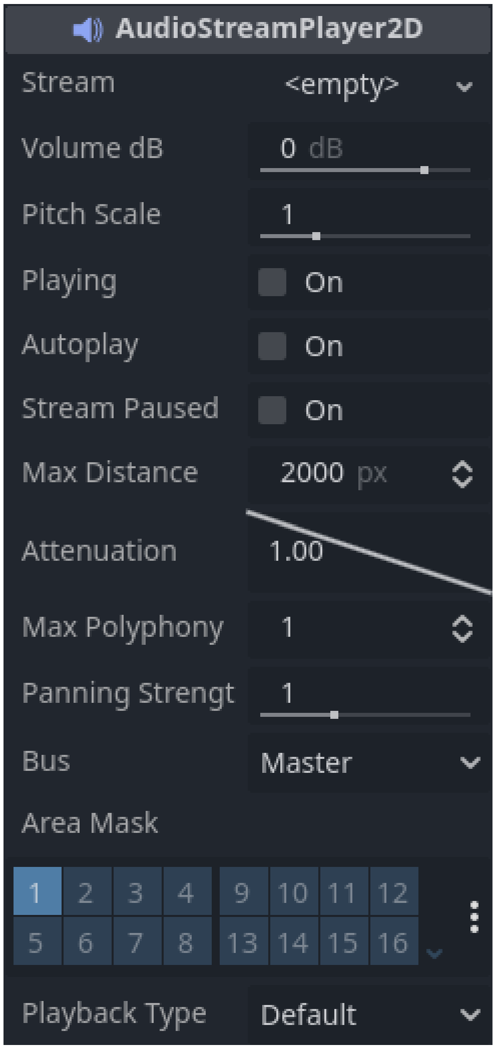
AudioStreamPlayer

Stream	<empty>
Volume dB	0 dB
Pitch Scale	1
Playing	☐ On
Autoplay	☐ On
Stream Paused	☐ On
Mix Target	Stereo
Max Polyphony	1
Bus	Master
Playback Type	Default

This is the standard, non-positional stream player. It can play to any bus. In 5.1 sound setups, it can send audio to stereo mix or front speakers.

Playback Type is an experimental setting, and could change in future versions of Godot. It exists so Web exports use Web Audio-API based samples instead of streaming all sounds to the browser, unlike most platforms. This prevents the audio from being garbled in single-threaded Web exports. By default, only the Web platform will use samples. Changing this setting is not recommended, unless you have an explicit reason to. You can change the default playback type for the web and other platforms in the project settings under **Audio > General** (advanced settings must be turned on to see the setting).

AudioStreamPlayer2D



AudioStreamPlayer2D

Stream <empty>

Volume dB 0 dB

Pitch Scale 1

Playing ☐ On

Autoplay ☐ On

Stream Paused ☐ On

Max Distance 2000 px

Attenuation 1.00

Max Polyphony 1

Panning Strengt 1

Bus Master

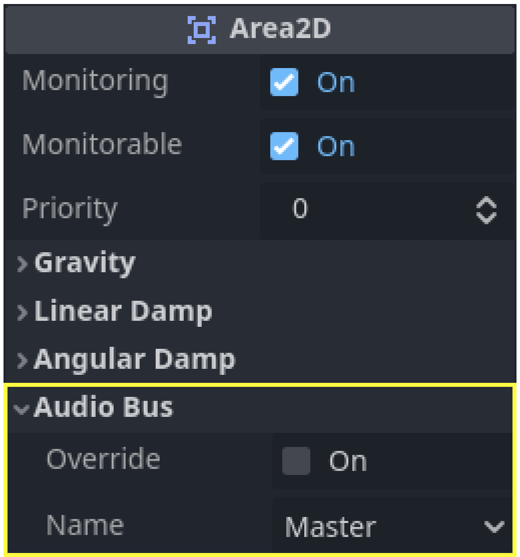
Area Mask

Playback Type Default

This is a variant of AudioStreamPlayer, but emits sound in a 2D positional environment. When close to the left of the screen, the panning will go left. When close to the right side, it will go right.

Note

Area2Ds can be used to divert sound from any AudioStreamPlayer2Ds they contain to specific buses. This makes it possible to create buses with different reverb or sound qualities to handle action happening in a particular parts of your game world.



Area2D

Monitoring ☑ On

Monitorable ☑ On

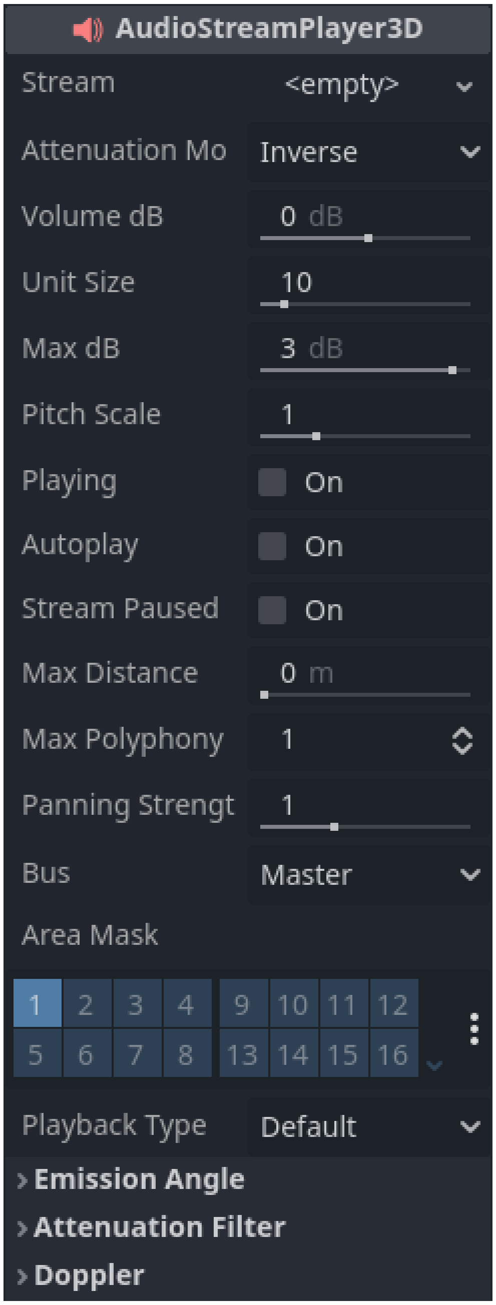
Priority 0

Gravity
Linear Damp
Angular Damp
Audio Bus

Override ☐ On

Name Master

AudioStreamPlayer3D



AudioStreamPlayer3D

Stream <empty>

Attenuation Mo Inverse

Volume dB 0 dB

Unit Size 10

Max dB 3 dB

Pitch Scale 1

Playing ☐ On

Autoplay ☐ On

Stream Paused ☐ On

Max Distance 0 m

Max Polyphony 1

Panning Strengt 1

Bus Master

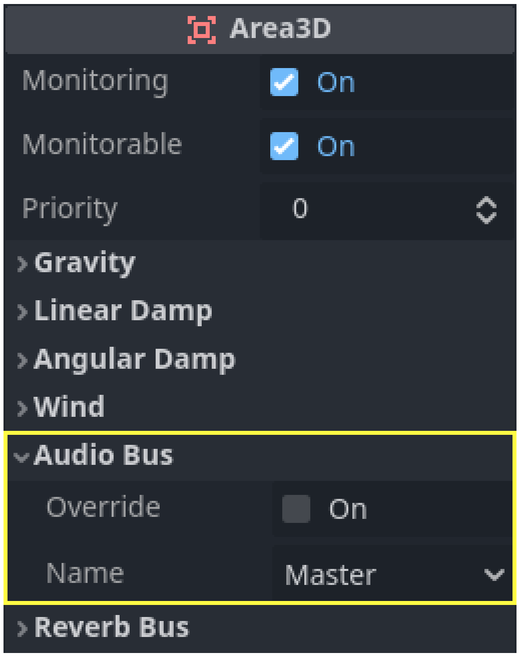
Area Mask

Playback Type Default

1
2
3
4
9
10
11
12
5
6
7
8
13
14
15
16
Emission Angle
Attenuation Filter
Doppler

This is a variant of AudioStreamPlayer, but emits sound in a 3D positional environment. Depending on the location of the player relative to the screen, it can position sound in stereo, 5.1 or 7.1 depending on the chosen audio setup.

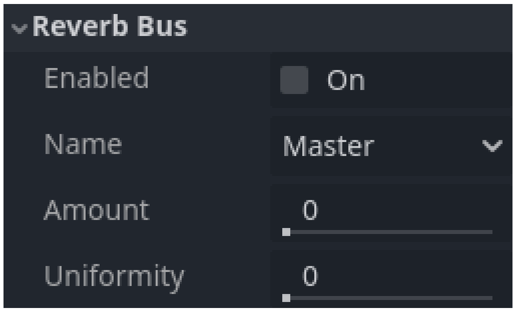
Similar to AudioStreamPlayer2D, an Area3D can divert the sound to an audio bus.


Area3D
Monitoring ☑ On
Monitorable ☑ On
Priority 0
Gravity
Linear Damp
Angular Damp
Wind
Audio Bus
Override ☐ On
Name Master
Reverb Bus

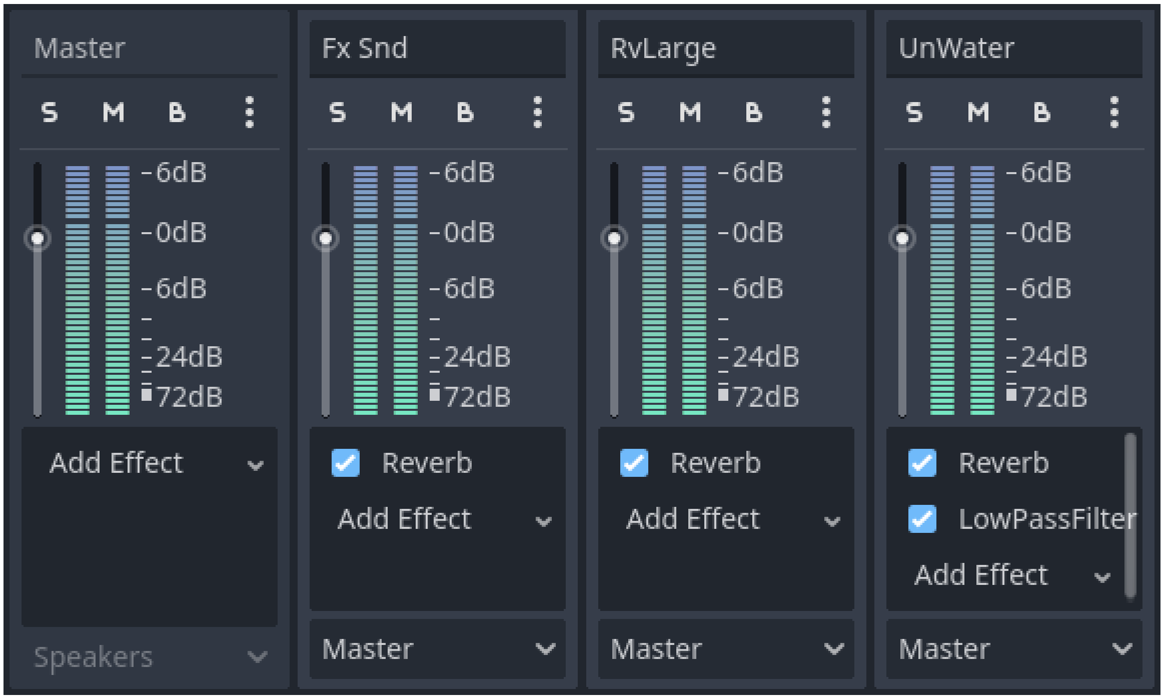
Unlike for 2D, the 3D version of AudioStreamPlayer has a few more advanced options:

Reverb buses

Godot allows for 3D audio streams that enter a specific Area3D node to send dry and wet audio to separate buses. This is useful when you have several reverb configurations for different types of rooms. This is done by enabling this type of reverb in the **Reverb Bus** section of the Area3D's properties:


Reverb Bus
Enabled ☐ On
Name Master
Amount 0
Uniformity 0

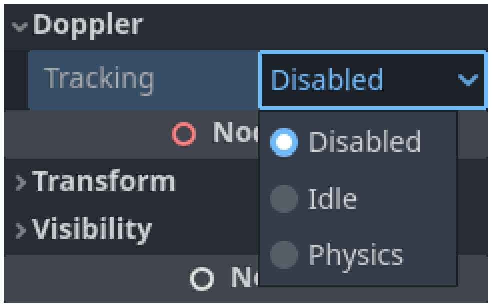
At the same time, a special bus layout is created where each Area3D receives the reverb info from each Area3D. A Reverb effect needs to be created and configured in each reverb bus to complete the setup for the desired effect:


Master
S M B
-6dB
-0dB
-6dB
-24dB
-72dB
Add Effect
Speakers
Fx Snd
S M B
☑ Reverb
Add Effect
Master
RvLarge
S M B
☑ Reverb
Add Effect
Master
UnWater
S M B
☑ Reverb
☑ LowPassFilter
Add Effect
Master

The Area3D's **Reverb Bus** section also has a parameter named **Uniformity**. Some types of rooms bounce sounds more than others (like a warehouse), so reverberation can be heard almost uniformly across the room even though the source may be far away. Playing around with this parameter can simulate that effect.

Doppler

When the relative velocity between an emitter and listener changes, this is perceived as an increase or decrease in the pitch of the emitted sound. Godot can track velocity changes in the AudioStreamPlayer3D and Camera nodes. Both nodes have this property, which must be enabled manually:



Tracking: Disabled

Doppler
Disabled
Idle
Physics
Transform
Visibility

Enable it by setting it depending on how objects will be moved: use **Idle** for objects moved using `_process`, or **Physics** for objects moved using `_physics_process`. The tracking will happen automatically.

Sync the gameplay with audio and music

Introduction

In any application or game, sound and music playback will have a slight delay. For games, this delay is often so small that it is negligible. Sound effects will come out a few milliseconds after any play() function is called. For music this does not matter as in most games it does not interact with the gameplay.

Still, for some games (mainly, rhythm games), it may be required to synchronize player actions with something happening in a song (usually in sync with the BPM). For this, having more precise timing information for an exact playback position is useful.

Achieving very low playback timing precision is difficult. This is because many factors are at play during audio playback:

- Audio is mixed in chunks (not continuously), depending on the size of audio buffers used (check latency in project settings).
- Mixed chunks of audio are not played immediately.
- Graphics APIs display two or three frames late.
- When playing on TVs, some delay may be added due to image processing.

The most common way to reduce latency is to shrink the audio buffers (again, by editing the latency setting in the project settings). The problem is that when latency is too small, sound mixing will require considerably more CPU. This increases the risk of skipping (a crack in sound because a mix callback was lost).

This is a common tradeoff, so Godot ships with sensible defaults that should not need to be altered.

The problem, in the end, is not this slight delay but synchronizing graphics and audio for games that require it. Beginning with Godot 3.2, some helpers were added to obtain more precise playback timing.

Using the system clock to sync

As mentioned before, If you call AudioStreamPlayer.play(), sound will not begin immediately, but when the audio thread processes the next chunk.

This delay can't be avoided but it can be estimated by calling AudioServer.get_time_to_next_mix().

The output latency (what happens after the mix) can also be estimated by calling AudioServer.get_output_latency().

Add these two and it's possible to guess almost exactly when sound or music will begin playing in the speakers during *_process()*:

GDScript

```
var time_begin
var time_delay


func _ready():
    time_begin = Time.get_ticks_usec()
    time_delay = AudioServer.get_time_to_next_mix() +
AudioServer.get_output_latency()
    $Player.play()


func _process(delta):
    # Obtain from ticks.
    var time = (Time.get_ticks_usec() - time_begin) / 1000000.0
    # Compensate for latency.
    time -= time_delay
    # May be below 0 (did not begin yet).
    time = max(0, time)
    print("Time is: ", time)
```

C#

```
private double _timeBegin;
private double _timeDelay;

public override void _Ready()
{
    _timeBegin = Time.GetTicksUsec();
    _timeDelay = AudioServer.GetTimeToNextMix() +
AudioServer.GetOutputLatency();
    GetNode<AudioStreamPlayer>("Player").Play();
}

public override void _Process(float _delta)
{
    double time = (Time.GetTicksUsec() - _timeBegin) /
1000000.0d;
    time = Math.Max(0.0d, time - _timeDelay);
    GD.Print(string.Format("Time is: {0}", time));
}
```

In the long run, though, as the sound hardware clock is never exactly in sync with the system clock, the timing information will slowly drift away.

For a rhythm game where a song begins and ends after a few minutes, this approach is fine (and it's the recommended approach). For a game where playback can last a much longer time, the game will eventually go out of sync and a different approach is needed.

Using the sound hardware clock to sync

Using AudioStreamPlayer.get_playback_position() to obtain the current position for the song sounds ideal, but it's not that useful as-is. This value will increment in chunks (every time the audio callback mixed a block of sound), so many calls can return the same value. Added to this, the value will be out of sync with the speakers too because of the previously mentioned reasons.

To compensate for the "chunked" output, there is a function that can help: AudioServer.get_time_since_last_mix().

Adding the return value from this function to *get_playback_position()* increases precision:

GDScript

```
var time = $Player.get_playback_position() +
AudioServer.get_time_since_last_mix()
```

C#

```
double time = GetNode<AudioStreamPlayer>
("Player").GetPlaybackPosition() +
AudioServer.GetTimeSinceLastMix();
```

To increase precision, subtract the latency information (how much it takes for the audio to be heard after it was mixed):

GDScript

```
var time = $Player.get_playback_position() +
AudioServer.get_time_since_last_mix() -
AudioServer.get_output_latency()
```

C#

```
double time = GetNode<AudioStreamPlayer>
("Player").GetPlaybackPosition() +
AudioServer.GetTimeSinceLastMix() -
AudioServer.GetOutputLatency();
```

The result may be a bit jittery due how multiple threads work. Just check that the value is not less than in the previous frame (discard it if so). This is also a less precise approach than the one before, but it will work for songs of any length, or synchronizing anything (sound effects, as an example) to music.

Here is the same code as before using this approach:

GDScript

```
func _ready():
    $Player.play()


func _process(delta):
    var time = $Player.get_playback_position() +
AudioServer.get_time_since_last_mix()
    # Compensate for output latency.
    time -= AudioServer.get_output_latency()
    print("Time is: ", time)
```

C#

```
public override void _Ready()
{
    GetNode<AudioStreamPlayer>("Player").Play();
}

public override void _Process(float _delta)
{
    double time = GetNode<AudioStreamPlayer>
("Player").GetPlaybackPosition() +
AudioServer.GetTimeSinceLastMix();
    // Compensate for output latency.
    time -= AudioServer.GetOutputLatency();
    GD.Print(string.Format("Time is: {0}", time));
}
```

Recording with microphone

Godot supports in-game audio recording for Windows, macOS, Linux, Android and iOS.

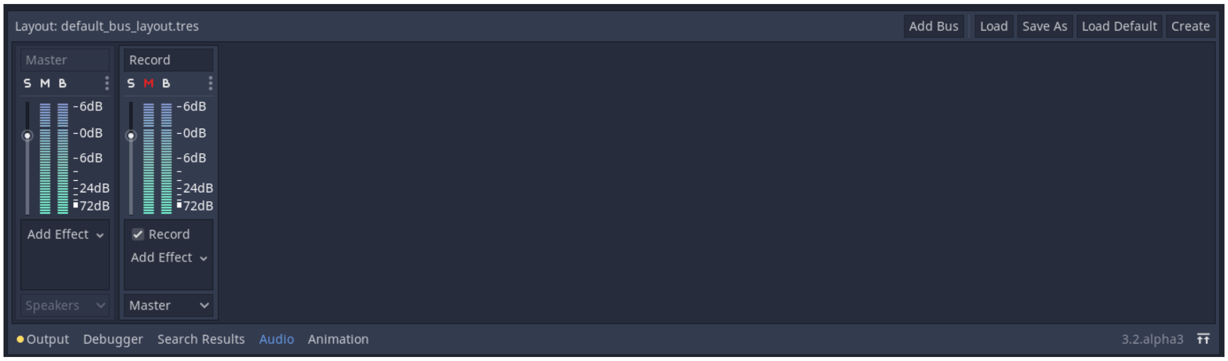
A simple demo is included in the official demo projects and will be used as support for this tutorial: https://github.com/godotengine/godot-demo-projects/tree/master/audio/mic_record.

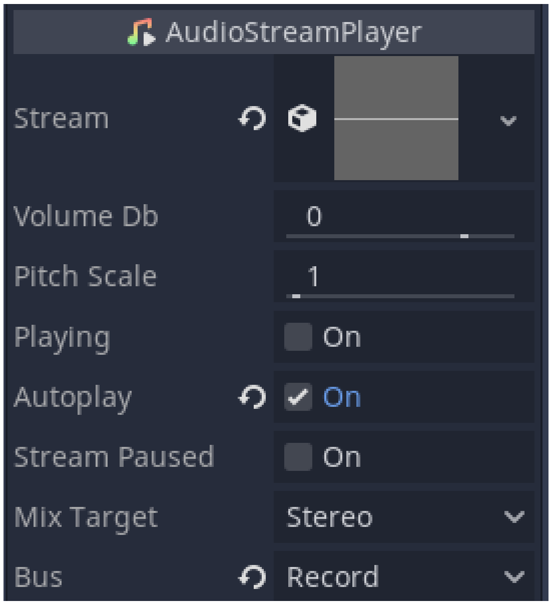
You will need to enable audio input in the project settings `Project Settings -> Audio -> Driver -> Enable Input`, or you'll just get empty audio files.

The structure of the demo

The demo consists of a single scene. This scene includes two major parts: the GUI and the audio.

We will focus on the audio part. In this demo, a bus named `Record` with the effect `Record` is created to handle the audio recording. An `AudioStreamPlayer` named `AudioStreamRecord` is used for recording.


Layout: default_bus_layout.tres
Add Bus
Load
Save As
Load Default
Create
Master
S M B
-6dB
-0dB
-6dB
-24dB
-72dB
Add Effect
Speakers
Record
S M B
-6dB
-0dB
-6dB
-24dB
-72dB
Record
Add Effect
Master
Output
Debugger
Search Results
Audio
Animation
3.2.alpha3


AudioStreamPlayer

Stream

Volume Db 0

Pitch Scale 1

Playing ☐ On

Autoplay ☑ On

Stream Paused ☐ On

Mix Target Stereo

Bus Record

GDScript

```
var effect
var recording


func _ready():
    # We get the index of the "Record" bus.
    var idx = AudioServer.get_bus_index("Record")
    # And use it to retrieve its first effect, which has been
defined
    # as an "AudioEffectRecord" resource.
    effect = AudioServer.get_bus_effect(idx, 0)
```

C#

```
private AudioEffectRecord _effect;
private AudioStreamSample _recording;

public override void _Ready()
{
    // We get the index of the "Record" bus.
    int idx = AudioServer.GetBusIndex("Record");
    // And use it to retrieve its first effect, which has been
defined
    // as an "AudioEffectRecord" resource.
    _effect = (AudioEffectRecord)AudioServer.GetBusEffect(idx,
0);
}
```

The audio recording is handled by the AudioEffectRecord resource which has three methods: get_recording(), is_recording_active(), and set_recording_active().

GDScript

```
func _on_record_button_pressed():
    if effect.is_recording_active():
        recording = effect.get_recording()
        $PlayButton.disabled = false
        $SaveButton.disabled = false
        effect.set_recording_active(false)
        $RecordButton.text = "Record"
        $Status.text = ""
    else:
        $PlayButton.disabled = true
        $SaveButton.disabled = true
        effect.set_recording_active(true)
        $RecordButton.text = "Stop"
        $Status.text = "Recording..."
```

C#

```
private void OnRecordButtonPressed()
{
    if (_effect.IsRecordingActive())
    {
        _recording = _effect.GetRecording();
        GetNode<Button>("PlayButton").Disabled = false;
        GetNode<Button>("SaveButton").Disabled = false;
        _effect.SetRecordingActive(false);
        GetNode<Button>("RecordButton").Text = "Record";
        GetNode<Label>("Status").Text = "";
    }
    else
    {
        GetNode<Button>("PlayButton").Disabled = true;
        GetNode<Button>("SaveButton").Disabled = true;
        _effect.SetRecordingActive(true);
        GetNode<Button>("RecordButton").Text = "Stop";
        GetNode<Label>("Status").Text = "Recording...";
    }
}
```

At the start of the demo, the recording effect is not active. When the user presses the `RecordButton`, the effect is enabled with `set_recording_active(true)`.

On the next button press, as `effect.is_recording_active()` is `true`, the recorded stream can be stored into the `recording` variable by calling `effect.get_recording()`.

GDScript

```
func _on_play_button_pressed():
    print(recording)
    print(recording.format)
    print(recording.mix_rate)
    print(recording.stereo)
    var data = recording.get_data()
    print(data.size())
    $AudioStreamPlayer.stream = recording
    $AudioStreamPlayer.play()
```

C#

```
private void OnPlayButtonPressed()
{
    GD.Print(_recording);
    GD.Print(_recording.Format);
    GD.Print(_recording.MixRate);
    GD.Print(_recording.Stereo);
    byte[] data = _recording.Data;
    GD.Print(data.Length);
    var audioStreamPlayer = GetNode<AudioStreamPlayer>
("AudioStreamPlayer");
    audioStreamPlayer.Stream = _recording;
    audioStreamPlayer.Play();
}
```

To playback the recording, you assign the recording as the stream of the `AudioStreamPlayer` and call `play()`.

GDScript

```
func _on_save_button_pressed():
    var save_path = $SaveButton/Filename.text
    recording.save_to_wav(save_path)
    $Status.text = "Saved WAV file to: %s\n(%s)" % [save_path,
ProjectSettings.globalize_path(save_path)]
```

C#

```
private void OnSaveButtonPressed()
{
    string savePath = GetNode<LineEdit>
("SaveButton/Filename").Text;
    _recording.SaveToWav(savePath);
    GetNode<Label>("Status").Text = string.Format("Saved WAV
file to: {0}\n({1})", savePath,
ProjectSettings.GlobalizePath(savePath));
}
```

To save the recording, you call `save_to_wav()` with the path to a file. In this demo, the path is defined by the user via a `LineEdit` input box.

Text to speech

Basic Usage

Basic usage of text-to-speech involves the following one-time steps:

- Enable TTS in the Godot editor for your project
- Query the system for a list of usable voices
- Store the ID of the voice you want to use

By default, the Godot project-level setting for text-to-speech is disabled, to avoid unnecessary overhead. To enable it:

- Go to **Project > Project Settings**
- Make sure the **Advanced Settings** toggle is enabled
- Click on **Audio > General**
- Ensure the **Text to Speech** option is checked
- Restart Godot if prompted to do so.

Text-to-speech uses a specific voice. Depending on the user's system, they might have multiple voices installed. Once you have the voice ID, you can use it to speak some text:

GDScript

```
# One-time steps.
# Pick a voice. Here, we arbitrarily pick the first English
voice.
var voices = DisplayServer.tts_get_voices_for_language("en")
var voice_id = voices[0]

# Say "Hello, world!".
DisplayServer.tts_speak("Hello, world!", voice_id)

# Say a longer sentence, and then interrupt it.
# Note that this method is asynchronous: execution proceeds to the next line immediately,
# before the voice finishes speaking.
var long_message = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur"
DisplayServer.tts_speak(long_message, voice_id)

# Immediately stop the current text mid-sentence and say goodbye instead.
DisplayServer.tts_stop()
DisplayServer.tts_speak("Goodbye!", voice_id)
```

C#

```
// One-time steps.
// Pick a voice. Here, we arbitrarily pick the first English voice.
string[] voices = DisplayServer.TtsGetVoicesForLanguage("en");
string voiceId = voices[0];

// Say "Hello, world!".
DisplayServer.TtsSpeak("Hello, world!", voiceId);

// Say a longer sentence, and then interrupt it.
// Note that this method is asynchronous: execution proceeds to the next line immediately,
// before the voice finishes speaking.
string longMessage = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur";
DisplayServer.TtsSpeak(longMessage, voiceId);

// Immediately stop the current text mid-sentence and say goodbye instead.
```

```
DisplayServer.TtsStop();
DisplayServer.TtsSpeak("Goodbye!", voiceId);
```

Requirements for functionality

Godot includes text-to-speech functionality. You can find these under the DisplayServer class.

Godot depends on system libraries for text-to-speech functionality. These libraries are installed by default on Windows and macOS, but not on all Linux distributions. If they are not present, text-to-speech functionality will not work. Specifically, the `tts_get_voices()` method will return an empty list, indicating that there are no usable voices.

Both Godot users on Linux and end-users on Linux running Godot games need to ensure that their system includes the system libraries for text-to-speech to work. Please consult the table below or your own distribution's documentation to determine what libraries you need to install.

Distro-specific one-liners

Arch Linux	`pacman -S speech-dispatcher festival espeakup`

Troubleshooting

If you get the error *Invalid get index '0' (on base: 'PackedStringArray').* for the line *var voice_id = voices[0]*, check if there are any items in *voices*. If not:

- All users: make sure you enabled **Text to Speech** in project settings
- Linux users: ensure you installed the system-specific libraries for text to speech

Best practices

The best practices for text-to-speech, in terms of the ideal player experience for blind players, is to send output to the player's screen reader. This preserves the choice of language, speed, pitch, etc. that the user set, as well as allows advanced features like allowing players to scroll backward and forward through text. As of now, Godot doesn't provide this level of integration.

With the current state of the Godot text-to-speech APIs, best practices include:

- Develop the game with text-to-speech enabled, and ensure that everything sounds correct
- Allow players to control which voice to use, and save/persist that selection across game sessions
- Allow players to control the speech rate, and save/persist that selection across game sessions

This provides your blind players with the most flexibility and comfort available when not using a screen reader, and minimizes the chance of frustrating and alienating them.

Caveats and Other Information

- Expect delays when you call *tts_speak* and *tts_stop*. The actual delay time varies depending on both the OS and on your machine's specifications. This is especially critical on Android and Web, where some of the voices depend on web services, and the actual time to playback depends on server load, network latency, and other factors.

- Non-English text works if the correct voices are installed and used. On Windows, you can consult the instructions in this article [https://www.ghacks.net/2018/08/11/unlock-all-windows-10-tts-voices-system-wide-to-get-more-of-them/] to enable additional language voices on Windows.
- Non-ASCII characters, such as umlaut, are pronounced correctly if you select the correct voice.
- Blind players use a number of screen readers, including JAWS, NVDA, VoiceOver, Narrator, and more.
- Windows text-to-speech APIs generally perform better than their equivalents on other systems (e.g. *tts_stop* followed by *tts_speak* immediately speaks the new message).

Export

See also

This section is about exporting a build of your project. If you're trying to export properties from a script, see GDScript exported properties or C# exported properties.

- Exporting projects
- Exporting packs, patches, and mods
- Feature tags
- Exporting for Linux
- Exporting for macOS
- Running Godot apps on macOS
- Exporting for Windows
- Changing application icon for Windows
- Exporting for iOS
- Exporting for Android
- Gradle builds for Android
- Exporting for the Web
- Exporting for dedicated servers
- One-click deploy

Exporting projects

Why export?

Originally, Godot did not have any means to export projects. The developers would compile the proper binaries and build the packages for each platform manually.

When more developers (and even non-programmers) started using it, and when our company started taking more projects at the same time, it became evident that this was a bottleneck.

On PC

Distributing a game project on PC with Godot is rather easy. Drop the Godot binary in the same directory as the `project.godot` file, then compress the project directory and you are done.

It sounds simple, but there are probably a few reasons why the developer may not want to do this. The first one is that it may not be desirable to distribute loads of files. Some developers may not like curious users peeking at how the game was made, others may find it inelegant, and so on. Another reason is that the developer might prefer a specially-compiled binary, which is smaller in size, more optimized and does not include tools like the editor and debugger.

Finally, Godot has a simple but efficient system for creating DLCs as extra package files.

On mobile

The same scenario on mobile platforms is a little worse. To distribute a project on those devices, a binary for each of those platforms is built, then added to a native project together with the game data.

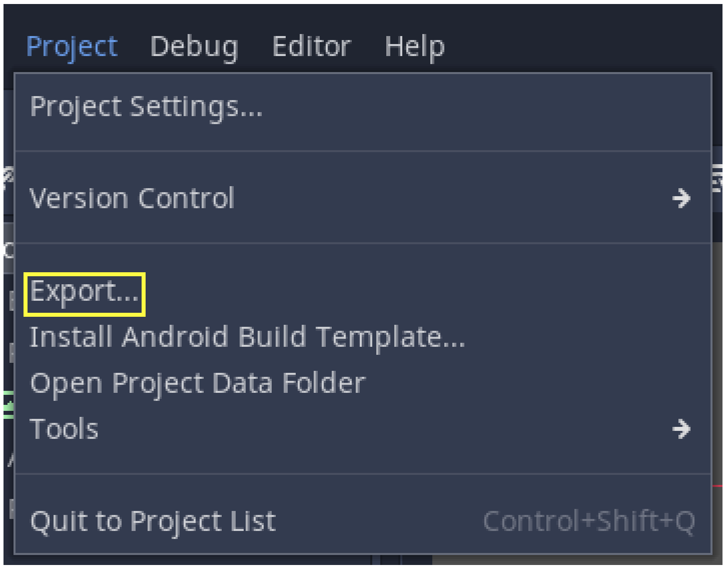
This can be troublesome because it means that the developer must be familiarized with the SDK of each platform before even being able to export. While learning each SDK is always encouraged, it can be frustrating to be forced to do it at an undesired time.

There is also another problem with this approach: different devices prefer some data in different formats to run. The main example of this is texture compression. All PC hardware uses S3TC (BC) compression and that has been standardized for more than a decade, but mobile devices use different formats for texture compression, such as ETC1 and ETC2.

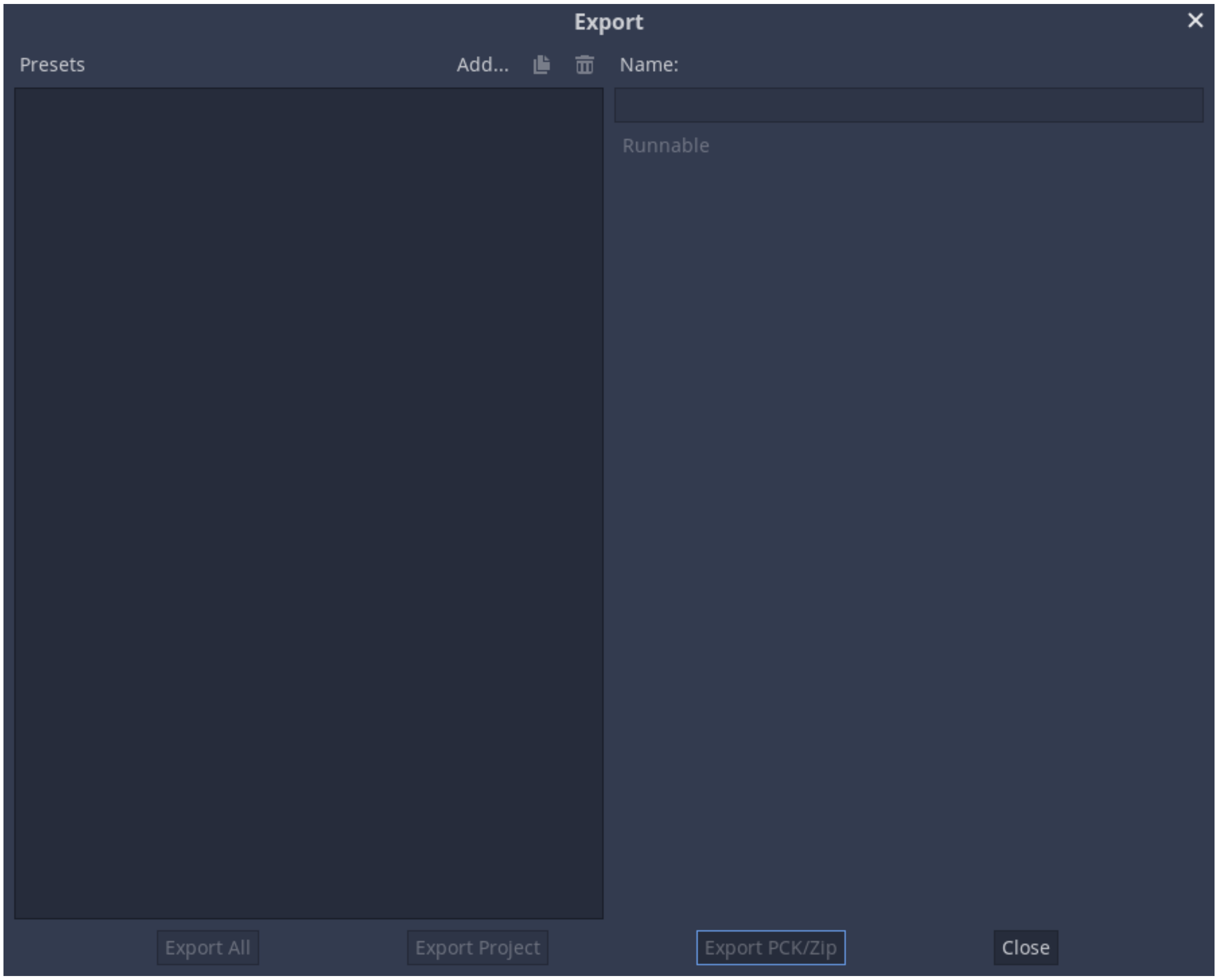
Export menu

After many attempts at different export workflows, the current one has proven to work the best. At the time of this writing, not all platforms are supported yet, but the supported platforms continue to grow.

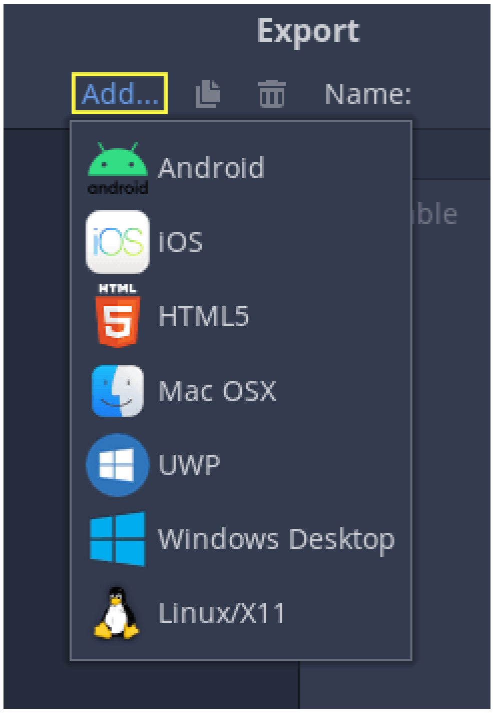
To open the export menu, click the **Export** button:


Project
Debug
Editor
Help
Project Settings...
Version Control
Export...
Install Android Build Template...
Open Project Data Folder
Tools
Quit to Project List
Control+Shift+Q

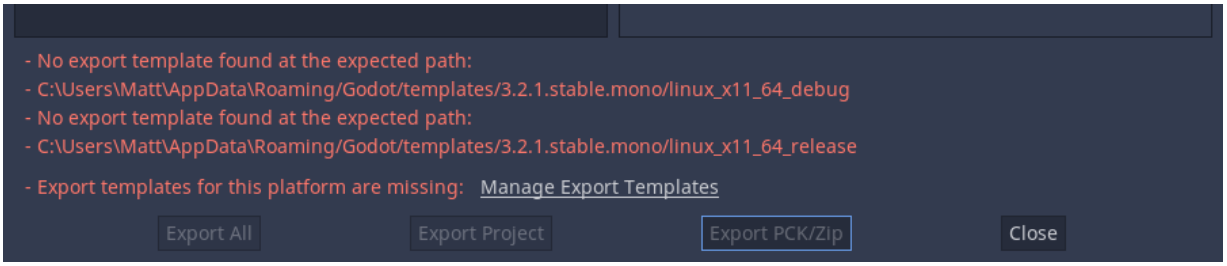
The export menu will open. However, it will be completely empty. This is because we need to add an export preset.


Export
Presets
Add...
Name:
Runnable
Export All
Export Project
Export PCK/Zip
Close

To create an export preset, click the **Add...** button at the top of the export menu. This will open a drop-down list of platforms to choose from for an export preset.


Export
Add...
Name:
Android
iOS
HTML5
Mac OSX
UWP
Windows Desktop
Linux/X11

The default options are often enough to export, so tweaking them is usually not necessary. However, many platforms require additional tools (SDKs) to be installed to be able to export. Additionally, Godot needs export templates installed to create packages. The export menu will complain when something is missing and will not allow the user to export for that platform until they resolve it:



```
- No export template found at the expected path:
- C:\Users\Matt\AppData\Roaming/Godot/templates/3.2.1.stable.mono/linux_x11_64_debug
- No export template found at the expected path:
- C:\Users\Matt\AppData\Roaming/Godot/templates/3.2.1.stable.mono/linux_x11_64_release
- Export templates for this platform are missing:  Manage Export Templates
```

Export All
Export Project
Export PCK/Zip
Close

At that time, the user is expected to come back to the documentation and follow instructions on how to properly set up that platform.

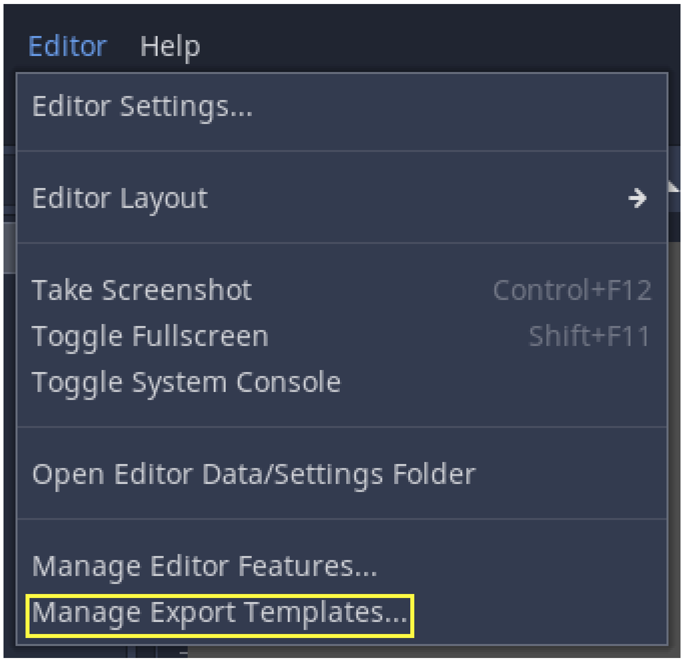
The buttons at the bottom of the menu allow you to export the project in a few different ways:

- Export All: Export the project as a playable build (Godot executable and project data) for all the presets defined. All presets must have an **Export Path** defined for this to work.
- Export Project: Export the project as a playable build (Godot executable and project data) for the selected preset.
- Export PCK/ZIP: Export the project resources as a PCK or ZIP package. This is not a playable build, it only exports the project data without a Godot executable.

Export templates

Apart from setting up the platform, the export templates must be installed to be able to export projects. They can be obtained as a TPZ file (which is a renamed ZIP archive) from the download page of the website [https://www.godotengine.org/download].

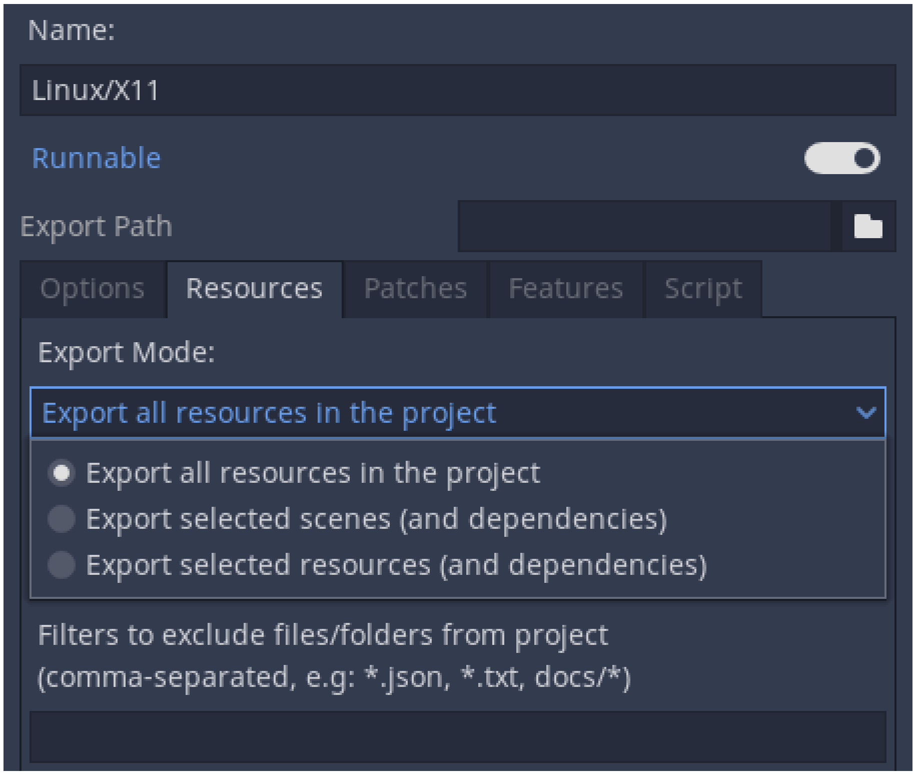
Once downloaded, they can be installed using the **Install Export Templates** option in the editor:


Editor
Help
Editor Settings...
Editor Layout
Take Screenshot
Control+F12
Toggle Fullscreen
Shift+F11
Toggle System Console
Open Editor Data/Settings Folder
Manage Editor Features...
Manage Export Templates...

Resource options

When exporting, Godot makes a list of all the files to export and then creates the package. There are 3 different modes for exporting:

- Export all resources in the project
- Export selected scenes (and dependencies)
- Export selected resources (and dependencies)



Name:

Linux/X11

Runnable

Export Path

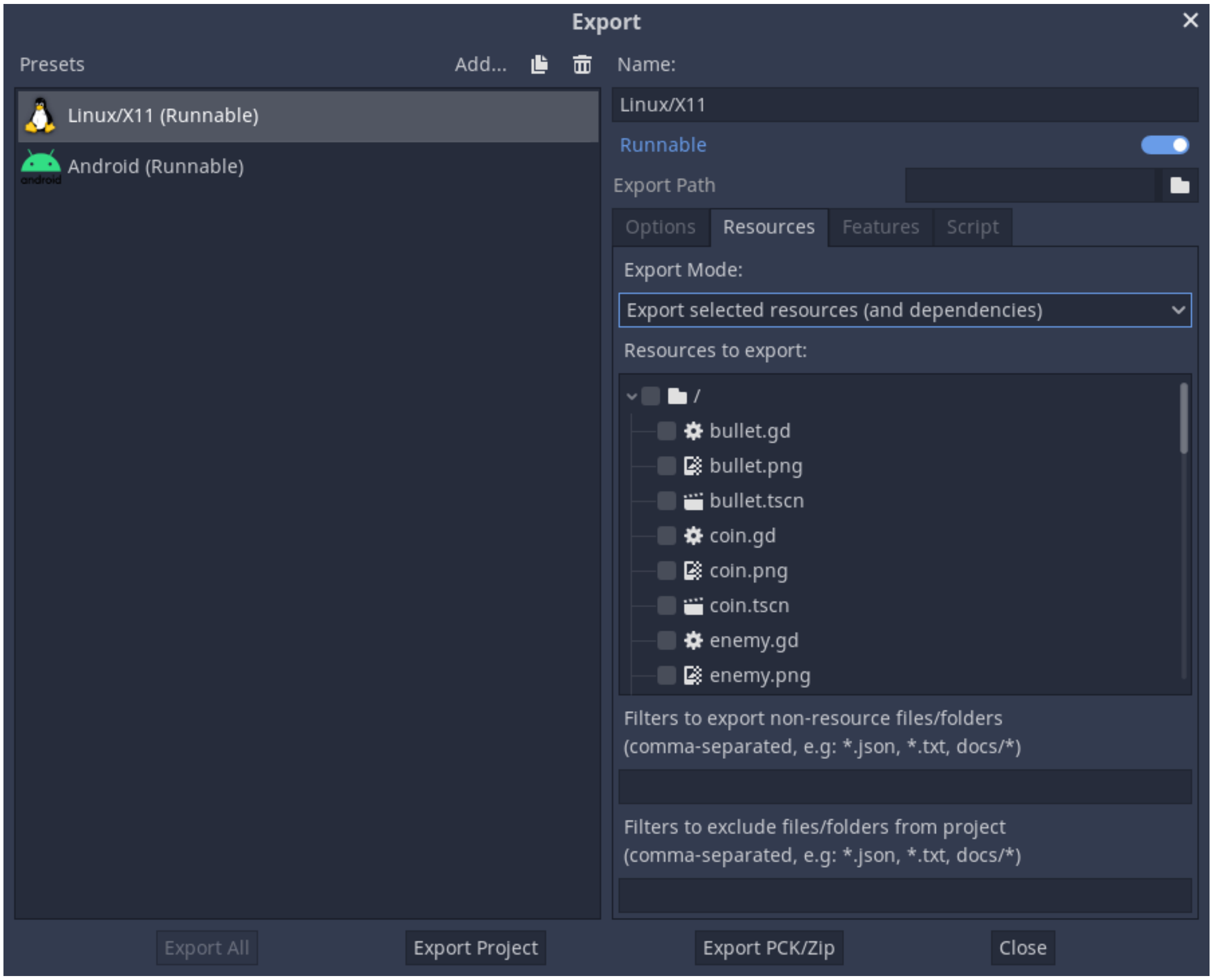
Export Mode:

Export all resources in the project

● Export all resources in the project
○ Export selected scenes (and dependencies)
○ Export selected resources (and dependencies)

Filters to exclude files/folders from project
(comma-separated, e.g: *.json, *.txt, docs/*)

Export all resources in the project will export every resource in the project. **Export selected scenes** and **Export selected resources** gives you a list of the scenes or resources in the project, and you have to select every scene or resource you want to export.



Name: Linux/X11

Runnable

Export Mode: Export selected resources (and dependencies)

Filters to export non-resource files/folders (comma-separated, e.g: *.json, *.txt, docs/*)

Filters to exclude files/folders from project (comma-separated, e.g: *.json, *.txt, docs/*)

Export
Presets
Add...
Linux/X11 (Runnable)
Android (Runnable)
Export Path
Options
Resources
Features
Script
Resources to export:
/
bullet.gd
bullet.png
bullet.tscn
coin.gd
coin.png
coin.tscn
enemy.gd
enemy.png
Export All
Export Project
Export PCK/Zip
Close

Note

Files and folders whose name begin with a period will never be included in the exported project. This is done to prevent version control folders like `.git` from being included in the exported PCK file.

Below the list of resources are two filters that can be setup. The first allows non resource files such as `.txt`, `.json` and `.csv` to be exported with the project. The second filter can be used to exclude every file of a certain type without manually deselecting every one. For example, `.png` files.

Configuration files

The export configuration is stored in two files that can both be found in the project directory:

- `export_presets.cfg`: This file contains the vast majority of the export configuration and can be safely committed to version control. There is nothing in here that you would normally have to keep secret.
- `.godot/export_credentials.cfg`: This file contains export options that are considered confidential, like passwords and encryption keys. It should generally **not** be committed to version control or shared with others unless you know exactly what you are doing.

Since the credentials file is usually kept out of version control systems, some export options will be missing if you clone the project to a new machine. The easiest way to deal with this is to copy the file manually from the old location to the new one.

Exporting from the command line

In production, it is useful to automate builds, and Godot supports this with the `--export-release` and `--export-debug` command line parameters. Exporting from the command line still requires an export preset to define the export parameters. A basic invocation of the command would be:

```
godot --export-release "Windows Desktop" some_name.exe
```

This will export to `some_name.exe`, assuming there is a preset called "Windows Desktop" and the template can be found. (The export preset name must be written within quotes if it contains spaces or special characters.) The output path is *relative to the project path* or *absolute*; **it does not respect the directory the command was invoked from**.

The output file extension should match the one used by the Godot export process:

- Windows: `.exe`
- macOS: `.zip` (from all platforms) or `.dmg` (only when exporting *from* macOS). `.app` is not supported directly, although the generated ZIP archive contains an `.app` bundle.
- Linux: Any extension (including none). `.x86_64` is typically used for 64-bit x86 binaries.
- HTML5: `.zip`
- Android: `.apk`
- iOS: `.zip`

You can also configure it to export *only* the PCK or ZIP file, allowing a single exported main pack file to be used with multiple Godot executables. When doing so, the export preset name must still be specified on the command line:

```
godot --export-pack "Windows Desktop" some_name.pck
```

It is often useful to combine the `--export-release` flag with the `--path` flag, so that you do not need to `cd` to the project folder before running the command:

```
godot --path /path/to/project --export-release "Windows
Desktop" some_name.exe
```

See also

See Command line tutorial for more information about using Godot from the command line.

PCK versus ZIP pack file formats

Each format has its upsides and downsides. PCK is the default and recommended format for most use cases, but you may want to use a ZIP archive instead depending on your needs.

PCK format:

- Uncompressed format. Larger file size, but faster to read/write.
- Not readable and writable using tools normally present on the user's operating system, even though there are third-party tools [https://github.com/hhyyrylainen/GodotPckTool] to extract and create PCK files.

ZIP format:

- Compressed format. Smaller file size, but slower to read/write.
- Readable and writable using tools normally present on the user's operating system. This can be useful to make modding easier (see also Exporting packs, patches, and mods).

Warning

Due to a known bug [https://github.com/godotengine/godot/pull/42123], when using a ZIP file as a pack file, the exported binary will not try to use it automatically. Therefore, you have to create a *launcher script* that the player can double-click or run from a terminal to launch the project:

```
:: launch.bat (Windows)
@echo off
my_project.exe --main-pack my_project.zip

# launch.sh (Linux)
./my_project.x86_64 --main-pack my_project.zip
```

Save the launcher script and place it in the same folder as the exported binary. On Linux, make sure to give executable permissions to the launcher script using the command `chmod +x launch.sh`.

Exporting packs, patches, and mods

Use cases

Oftentimes one would like to add functionality to one's game after it has been deployed.

Examples of this include...

- Downloadable Content: the ability to add features and content to one's game.
- Patches: the ability to fix a bug that is present in a shipped product.
- Mods: grant other people the ability to create content for one's game.

These tools help developers to extend their development beyond the initial release.

Overview of PCK files

Godot enables this via a feature called **resource packs** (PCK files, with extension `.pck`).

Advantages:

- incremental updates/patches
- offer DLCs
- offer mod support
- no source code disclosure needed for mods

- more modular project structure
- users don't have to replace the entire game

The first part of using them involves exporting and delivering the project to players. Then, when one wants to add functionality or content later on, they just deliver the updates via PCK files to the users.

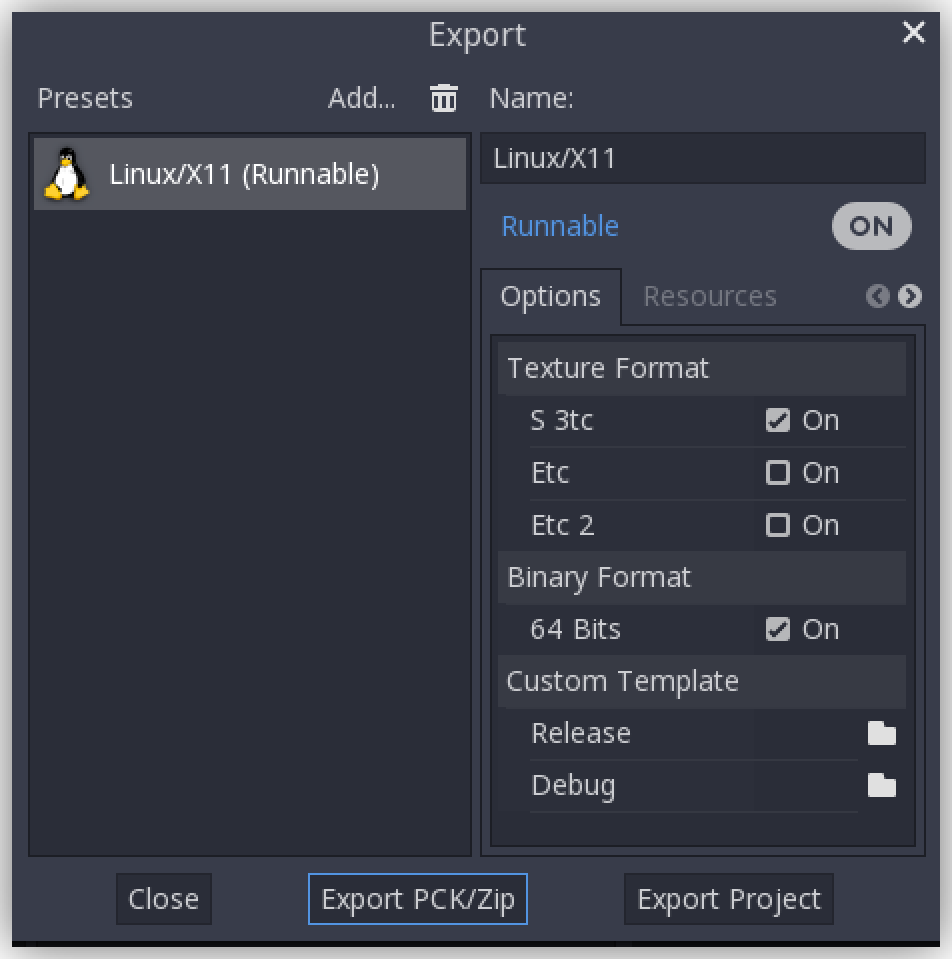
PCK files usually contain, but are not limited to:

- scripts
- scenes
- shaders
- models
- textures
- sound effects
- music
- any other asset suitable for import into the game

The PCK files can even be an entirely different Godot project, which the original game loads in at runtime.

Generating PCK files

In order to pack all resources of a project into a PCK file open the project and go to Project/Export and click on "Export PCK/Zip". Also make sure to have an export template selected while doing so.


Export
Presets
Add...
Linux/X11 (Runnable)
Options
Resources
Close
Export PCK/Zip
Export Project

Name: Linux/X11

Runnable ON

Texture Format

S 3tc ☑ On

Etc ☐ On

Etc 2 ☐ On

Binary Format

64 Bits ☑ On

Custom Template

Release

Debug

Another method would be to [export from the command line](). If the output file ends with a PCK or ZIP file extension, then the export process will build that type of file for the chosen platform.

Note

If one wishes to support mods for their game, they will need their users to create similarly exported files. Assuming the original game expects a certain structure for the PCK's resources and/or a certain interface for its scripts, then either...

1. The developer must publicize documentation of these expected structures/ interfaces, expect modders to install Godot Engine, and then also expect those modders to conform to the documentation's defined API when building mod content for the game (so that it will work). Users would then use Godot's built in exporting tools to create a PCK file, as detailed above.
2. The developer uses Godot to build a GUI tool for adding their exact API content to a project. This Godot tool must either run on a tools-enabled build of the engine or have access to one (distributed alongside or perhaps in the original game's files). The tool can then use the Godot executable to export a PCK file from the command line with OS.execute(). The game itself shouldn't use a tool-build of the engine (for security), so it's best to keep the modding tool and game separate.

Opening PCK files at runtime

To import a PCK file, one uses the ProjectSettings singleton. The following example expects a “mod.pck” file in the directory of the games executable. The PCK file contains a “mod_scene.tscn” test scene in its root.

GDScript

```
func _your_function():
    # This could fail if, for example, mod.pck cannot be found.
    var success =
ProjectSettings.load_resource_pack("res://mod.pck")

    if success:
        # Now one can use the assets as if they had them in the
project from the start.
        var imported_scene = load("res://mod_scene.tscn")
```

C#

```
private void YourFunction()
{
    // This could fail if, for example, mod.pck cannot be
found.
    var success =
ProjectSettings.LoadResourcePack("res://mod.pck");

    if (success)
    {
        // Now one can use the assets as if they had them in
the project from the start.
        var importedScene =
(PackedScene)ResourceLoader.Load("res://mod_scene.tscn");
    }
}
```

Warning

By default, if you import a file with the same file path/name as one you already have in your project, the imported one will replace it. This is something to watch out for when creating DLC or mods. You can solve this problem by using a tool that isolates mods to a specific mods subfolder. However, it is also a way of creating patches for one's own game. A PCK file of this kind can fix the content of a previously loaded PCK.

To opt out of this behavior, pass `false` as the second argument to ProjectSettings.load_resource_pack().

Note

For a C# project, you need to build the DLL and place it in the project directory first. Then, before loading the resource pack, you need to load its DLL as follows: `Assembly.LoadFile("mod.dll")`

Summary

This tutorial explains how to add mods, patches, or DLC to a game. The most important thing is to identify how one plans to distribute future content for their game and develop a workflow that is customized for that purpose. Godot should make that process smooth regardless of which route a developer pursues.

Feature tags

Introduction

Godot has a special system to tag availability of features. Each *feature* is represented as a string, which can refer to many of the following:

- Platform name.
- Platform architecture (64-bit or 32-bit, x86 or ARM).
- Platform type (desktop, mobile, Web).
- Supported texture compression algorithms on the platform.
- Whether a build is `debug` or `release` (`debug` includes the editor).
- Whether the project is running from the editor or a "standalone" binary.
- Many more things.

Features can be queried at run-time from the singleton API by calling:

GDScript

```
OS.has_feature(name)
```

C#

```
OS.HasFeature(name);
```

OS feature tags are used by GDExtension to determine which libraries to load. For example, a library for `linux.debug.editor.x86_64` will be loaded only on a debug editor build for Linux x86_64.

Default features

Here is a list of most feature tags in Godot. Keep in mind they are **case-sensitive**:

Feature tag	Description
android	Running on Android (but not within a Web browser)
bsd	Running on *BSD (but not within a Web browser)
linux	Running on Linux (but not within a Web browser)
macos	Running on macOS (but not within a Web browser)
ios	Running on iOS (but not within a Web browser)
windows	Running on Windows
linuxbsd	Running on Linux or *BSD
debug	Running on a debug build (including the editor)
release	Running on a release build

Feature tag	Description
editor	Running on an editor build
template	Running on a non-editor (export template) build
double	Running on a double-precision build
single	Running on a single-precision build
64	Running on a 64-bit build (any architecture)
32	Running on a 32-bit build (any architecture)
x86_64	Running on a 64-bit x86 build
x86_32	Running on a 32-bit x86 build
x86	Running on an x86 build (any bitness)
arm64	Running on a 64-bit ARM build
arm32	Running on a 32-bit ARM build
arm	Running on an ARM build (any bitness)

Feature tag	Description
rv64	Running on a 64-bit RISC-V build
riscv	Running on a RISC-V build (any bitness)
ppc64	Running on a 64-bit PowerPC build
ppc32	Running on a 32-bit PowerPC build
ppc	Running on a PowerPC build (any bitness)
wasm64	Running on a 64-bit WebAssembly build (not yet possible)
wasm32	Running on a 32-bit WebAssembly build
wasm	Running on a WebAssembly build (any bitness)
mobile	Host OS is a mobile platform
pc	Host OS is a PC platform (desktop/laptop)
web	Host OS is a Web browser

Feature tag	Description
web_android	Host OS is a Web browser running on Android
web_ios	Host OS is a Web browser running on iOS
web_linuxbsd	Host OS is a Web browser running on Linux or *BSD
web_macos	Host OS is a Web browser running on macOS
web_windows	Host OS is a Web browser running on Windows
etc	Textures using ETC1 compression are supported
etc2	Textures using ETC2 compression are supported
s3tc	Textures using S3TC (DXT/BC) compression are supported
movie	Movie Maker mode is active

Warning

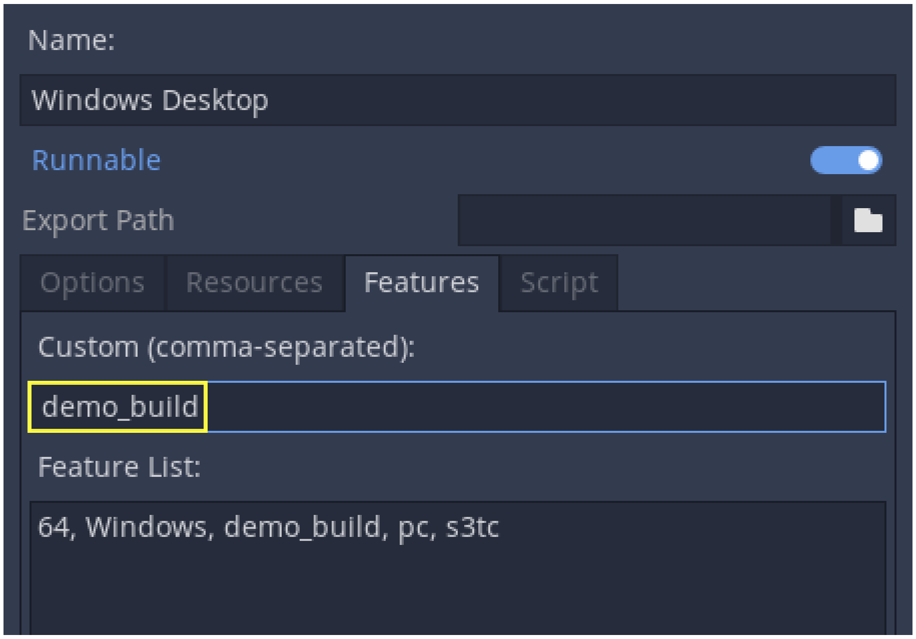
With the exception of texture compression, `web_<platform>` and `movie` feature tags, default feature tags are **immutable**. This means that they will *not* change depending on run-time conditions.

For example, `OS.has_feature("mobile")` will return `false` when running a project exported to Web on a mobile device.

To check whether a project exported to Web is running on a mobile device, use `OS.has_feature("web_android") or OS.has_feature("web_ios")`.

Custom features

It is possible to add custom features to a build; use the relevant field in the *export preset* used to generate it:



Name: Windows Desktop

Runnable

Export Path

Custom (comma-separated): demo_build

Feature List: 64, Windows, demo_build, pc, s3tc

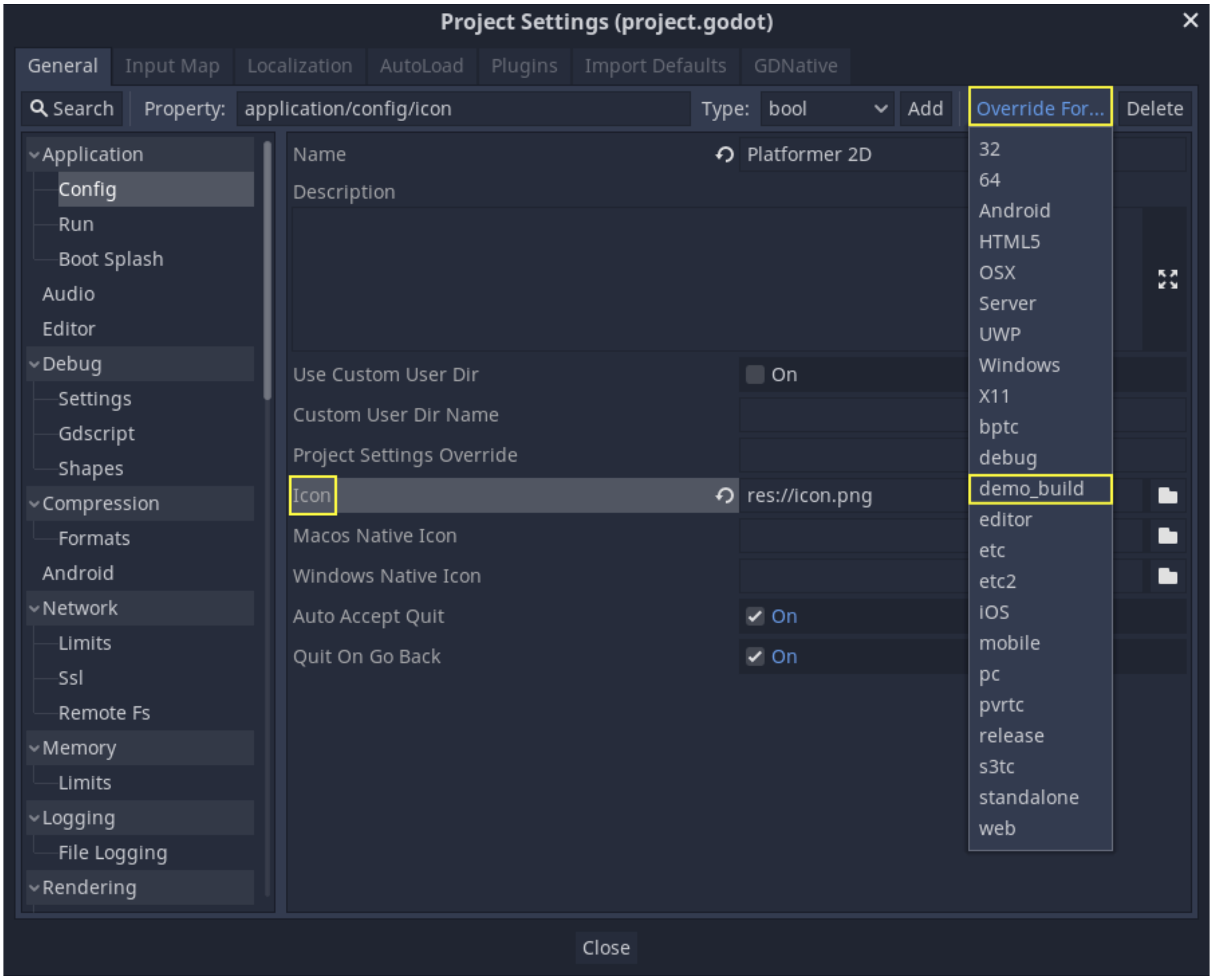
Note

Custom feature tags are only used when running the exported project (including with One-click deploy). They are **not used** when running the project from the editor, even if the export preset marked as **Runnable** for your current platform has custom feature tags defined.

Overriding project settings

Features can be used to override specific configuration values in the *Project Settings*. This allows you to better customize any configuration when doing a build.

In the following example, a different icon is added for the demo build of the game (which was customized in a special export preset, which, in turn, includes only demo levels).



Project Settings (project.godot)

Property: application/config/icon

Type: bool

Name: Platformer 2D

Description

Use Custom User Dir: ☐ On

Custom User Dir Name

Project Settings Override

Icon: res://icon.png

Macos Native Icon

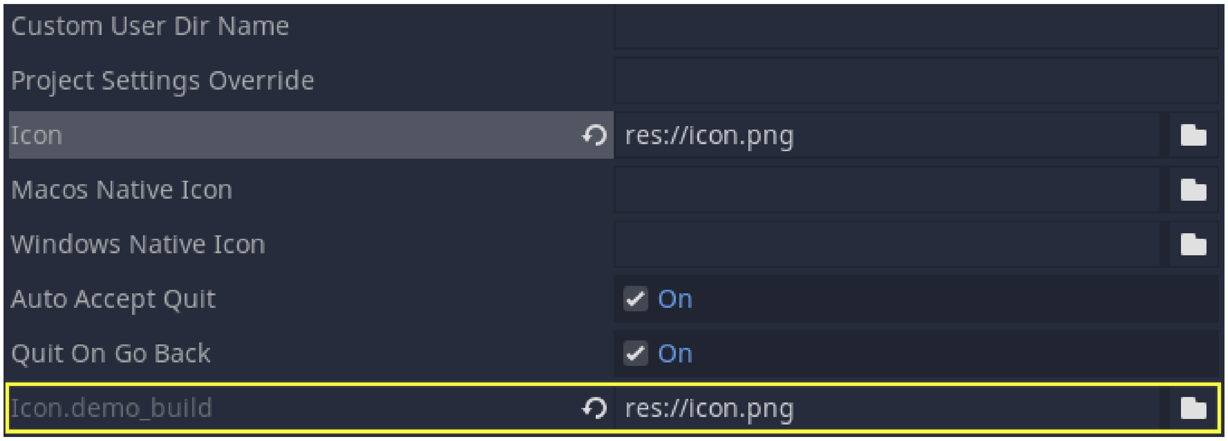
Windows Native Icon

Auto Accept Quit: ☑ On

Quit On Go Back: ☑ On

General
Input Map
Localization
AutoLoad
Plugins
Import Defaults
GDNative
Search
Add
Override For...
Delete
Application
Config
Run
Boot Splash
Audio
Editor
Debug
Settings
Gdscript
Shapes
Compression
Formats
Android
Network
Limits
Ssl
Remote Fs
Memory
Limits
Logging
File Logging
Rendering
32
64
Android
HTML5
OSX
Server
UWP
Windows
X11
bptc
debug
demo_build
editor
etc
etc2
iOS
mobile
pc
pvrtc
release
s3tc
standalone
web
Close

After overriding, a new field is added for this specific configuration:


Custom User Dir Name

Project Settings Override

Icon: res://icon.png

Macos Native Icon

Windows Native Icon

Auto Accept Quit: ☑ On

Quit On Go Back: ☑ On

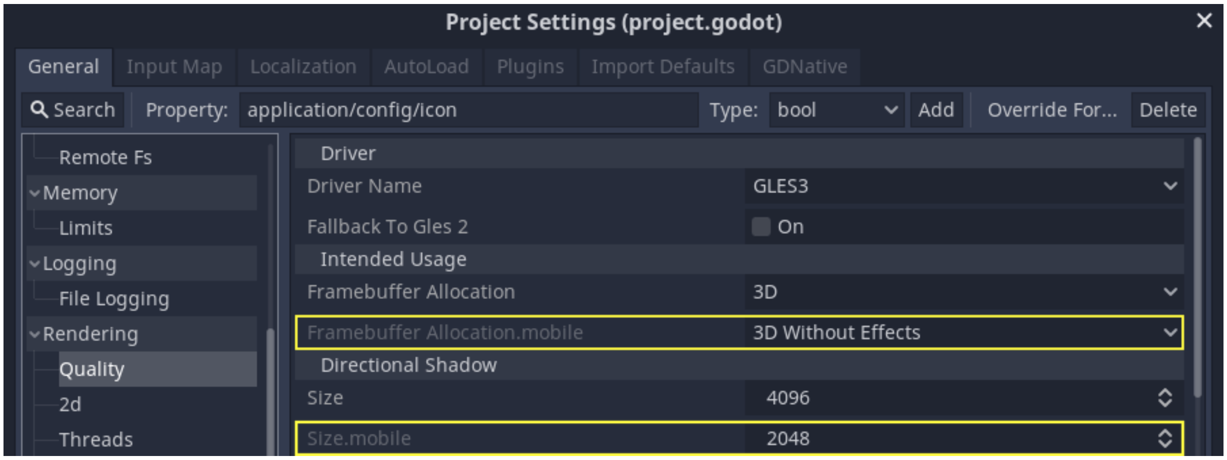
Icon.demo_build: res://icon.png

Note

When using the project settings "override.cfg" functionality (which is unrelated to feature tags), remember that feature tags still apply. Therefore, make sure to *also* override the setting with the desired feature tag(s) if you want them to override base project settings on all platforms and configurations.

Default overrides

There are already a lot of settings that come with overrides by default; they can be found in many sections of the project settings.


Project Settings (project.godot)
General
Input Map
Localization
AutoLoad
Plugins
Import Defaults
GDNative
Search
Add
Override For...
Delete
Remote Fs
Memory
Limits
Logging
File Logging
Rendering
Quality
2d
Threads
Property: application/config/icon

Type: bool

Driver

Driver Name: GLES3

Fallback To Gles 2: ☐ On

Intended Usage

Framebuffer Allocation: 3D

Framebuffer Allocation.mobile: 3D Without Effects

Directional Shadow

Size: 4096

Size.mobile: 2048

Customizing the build

Feature tags can be used to customize a build process too, by writing a custom **ExportPlugin**. They are also used to specify which shared library is loaded and exported in **GDExtension**.

Exporting for Linux

See also

This page describes how to export a Godot project to Linux. If you're looking to compile editor or export template binaries from source instead, read Compiling for Linux, *BSD.

The simplest way to distribute a game for PC is to copy the executable (`godot`), compress the folder and send it to someone else. However, this is often not desired.

Godot offers a more elegant approach for PC distribution when using the export system. When exporting for Linux, the exporter takes all the project files and creates a `data.pck` file. This file is bundled with a specially optimized binary that is smaller, faster and does not contain the editor and debugger.

Environment variables

You can use the following environment variables to set export options outside of the editor. During the export process, these override the values that you set in the export menu.

Linux export environment variables

Export option	Environment variable
Encryption / Encryption Key	`GODOT_SCRIPT_ENCRYPTION_KEY`

Exporting for macOS

See also

This page describes how to export a Godot project to macOS. If you're looking to compile editor or export template binaries from source instead, read Compiling for macOS.

macOS apps exported with the official export templates are exported as a single "Universal 2" binary `.app` bundle, a folder with a specific structure which stores the executable, libraries and all the project files. This bundle can be exported as is, packed in a ZIP archive or DMG disk image (only supported when exporting from a computer running macOS). Universal binaries for macOS support both Intel x86_64 and ARM64 (Apple silicon, i.e. M1) architectures [https://developer.apple.com/documentation/apple-silicon/building-a-universal-macos-binary].

Requirements

- Download the Godot export templates. Use the Godot menu: `Editor > Manage Export Templates`.
- A valid and unique `Bundle identifier` should be set in the `Application` section of the export options.

Warning

Projects exported without code signing and notarization will be blocked by Gatekeeper if they are downloaded from unknown sources, see the Running Godot apps on macOS page for more information.

Code signing and notarization

By default, macOS will run only applications that are signed and notarized. If you use any other signing configuration, see Running Godot apps on macOS for workarounds.

To notarize an app, you **must** have a valid Apple Developer ID Certificate [https://developer.apple.com/].

If you have an Apple Developer ID Certificate and exporting from macOS

Install Xcode [https://developer.apple.com/xcode/] command line tools and open Xcode at least once or run the `sudo xcodebuild -license accept` command to accept license agreement.

To sign exported app

- Select `Xcode codesign` in the `Code Signing > Codesign` option.
- Set valid Apple ID certificate identity (certificate "Common Name") in the `Code Signing > Identity` section.

To notarize exported app

- Select `Xcode altool` in the `Notarization > Notarization` option.
- Disable the `Debugging` entitlement.
- Set valid Apple ID login / app. specific password or App Store Connect [https://developer.apple.com/documentation/appstoreconnectapi] API UUID / Key in the `Notarization` section.

You can use the `xcrun notarytool history` command to check notarization status and use the `xcrun notarytool log {ID}` command to download the notarization log.

If you encounter notarization issues, see Resolving common notarization issues [https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution/resolving_common_notarization_issues] for more info.

After notarization is completed, staple the ticket [https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution/customizing_the_notarization_workflow] to the exported project.

If you have an Apple Developer ID Certificate and exporting from Linux or Windows

Install PyOxidizer rcodesign [https://github.com/indygreg/apple-platform-rs/tree/main/apple-codesign], and configure the path to `rcodesign` in the `Editor Settings > Export > macOS > rcodesign`.

To sign exported app

- Select `PyOxidizer rcodesign` in the `Code Signing > Codesign` option.
- Set valid Apple ID PKCS #12 certificate file and password in the `Code Signing` section.

To notarize exported app

- Select `PyOxidizer rcodesign` in the `Notarization > Notarization` option.
- Disable the `Debugging` entitlement.

- Set valid App Store Connect [https://developer.apple.com/documentation/appstoreconnectapi] API UUID / Key in the `Notarization` section.

You can use the `rcodesign notary-log` command to check notarization status.

After notarization is completed, use the `rcodesign staple` command to staple the ticket to the exported project.

If you do not have an Apple Developer ID Certificate

- Select `Built-in (ad-hoc only)` in the `Code Signing > Codesign` option.
- Select `Disabled` in the `Notarization > Notarization` option.

In this case Godot will use an ad-hoc signature, which will make running an exported app easier for the end users, see the Running Godot apps on macOS page for more information.

Signing Options

Option	Description
Codesign	Tool to use for code signing.
Identity	The "Full Name" or "Common Name" of the signing identity, store in the macOS keychain. [1]

Option	**Description**
Certificate File	The PKCS #12 certificate file. [2]
Certificate Password	Password for the certificate file. [2]
Custom Options	Array of command line arguments passed to the code signing tool.

[1] This option is visible only when signing with Xcode codesign.

[2] (*1*,*2*) These options are visible only when signing with PyOxidizer rcodesign.

Notarization Options

Option	**Description**
Notarization	Tool to use for notarization.
Apple ID Name	Apple ID account name (email address). [3]
Apple ID Password	Apple ID app-specific password. See Using app-specific passwords to enable two-factor authentication and create app password. [3]

Option	Description
Apple Team ID	Team ID ("Organization Unit"), if your Apple ID belongs to multiple teams (optional). [3]
API UUID	Apple App Store Connect API issuer UUID.
API Key	Apple App Store Connect API key.

Note

You should set either Apple ID Name/Password or App Store Connect API UUID/Key.

[3] (*1*,*2*,*3*) These options are visible only when notarizing with Xcode altool.

See Notarizing macOS Software Before Distribution [https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution?language=objc] for more info.

Entitlements

Hardened Runtime Entitlements

Hardened Runtime entitlements manage security options and resource access policy. See Hardened Runtime [https://developer.apple.com/documentation/security/hardened_runtime?language=objc] for more info.

Entitlement	**Description**
Allow JIT Code Execution [4]	Allows creating writable and executable memory for JIT code. If you are using add-ons with dynamic or self-modifying native code, enable them according to the add-on documentation.
Allow Unsigned Executable Memory [4]	Allows creating writable and executable memory without JIT restrictions. If you are using add-ons with dynamic or self-modifying native code, enable them according to the add-on documentation.
Allow DYLD Environment Variables [4]	Allows app to uss dynamic linker environment variables to inject code. If you are using add-ons with dynamic or self-modifying native code, enable them according to the add-on documentation.
Disable Library Validation	Allows app to load arbitrary libraries and frameworks. Enable it if you are using GDExtension add-ons or ad-hoc signing, or want to support user-provided external add-ons.
Audio Input	Enable if you need to use the microphone or other audio input sources, if it's enabled you should also provide usage message in the *privacy/microphone_usage_description* option.
Camera	Enable if you need to use the camera, if it's enabled you should also provide usage message in the *privacy/camera_usage_description* option.

Entitlement	Description
Location	Enable if you need to use location information from Location Services, if it's enabled you should also provide usage message in the *privacy/location_usage_description* option.
Address Book	[5] Enable to allow access contacts in the user's address book, if it's enabled you should also provide usage message in the *privacy/address_book_usage_description* option.
Calendars	[5] Enable to allow access to the user's calendar, if it's enabled you should also provide usage message in the *privacy/calendar_usage_description* option.
Photo Library	[5] Enable to allow access to the user's Photos library, if it's enabled you should also provide usage message in the *privacy/photos_library_usage_description* option.
Apple Events	[5] Enable to allow app to send Apple events to other apps.
Debugging	[6] You can temporarily enable this entitlement to use native debugger (GDB, LLDB) with the exported app. This entitlement should be disabled for production export.

[4] (*1*,*2*,*3*) The `Allow JIT Code Execution`, `Allow Unsigned Executable Memory` and `Allow DYLD Environment Variables` entitlements are always enabled for the Godot Mono exports, and are not visible in the export options.

[5] (*1*,*2*,*3*,*4*) These features aren't supported by Godot out of the box, enable them only if you are using add-ons which require them.

[6] To notarize an app, you must disable the `Debugging` entitlement.

App Sandbox Entitlement

The App Sandbox restricts access to user data, networking and devices. Sandboxed apps can't access most of the file system, can't use custom file dialogs and execute binaries (using `OS.execute` and `OS.create_process`) outside the `.app` bundle. See App Sandbox [https://developer.apple.com/documentation/security/app_sandbox?language=objc] for more info.

Note

To distribute an app through the App Store, you must enable the App Sandbox.

Entitlement	Description
Enabled	Enables App Sandbox.
Network Server	Enable to allow app to listen for incoming network connections.

Entitlement	**Description**
Network Client	Enable to allow app to establish outgoing network connections.
Device USB	Enable to allow app to interact with USB devices. This entitlement is required to use wired controllers.
Device Bluetooth	Enable to allow app to interact with Bluetooth devices. This entitlement is required to use wireless controllers.
Files Downloads [7]	Allows read or write access to the user's "Downloads" folder.
Files Pictures [7]	Allows read or write access to the user's "Pictures" folder.
Files Music [7]	Allows read or write access to the user's "Music" folder.
Files Movies [7]	Allows read or write access to the user's "Movies" folder.
Files User Selected [7]	Allows read or write access to arbitrary folder. To gain access, a folder must be selected from the native file dialog by the user.

Entitlement	Description
Helper Executable	List of helper executables to embedded to the app bundle. Sandboxed app are limited to execute only these executable.

[7] (*1*,*2*,*3*,*4*,*5*) You can optionally provide usage messages for various folders in the *privacy/*_folder_usage_description* options.

Note

You can override default entitlements by selecting custom entitlements file, in this case all other entitlement are ignored.

Environment variables

You can use the following environment variables to set export options outside of the editor. During the export process, these override the values that you set in the export menu.

macOS export environment variables

Export option	Environment variable
Encryption / Encryption Key	`GODOT_SCRIPT_ENCRYPTION_KEY`
Options / Codesign / Certificate File	`GODOT_MACOS_CODESIGN_CERTIFICATE_FILE`

Export option	Environment variable
Options / Codesign / Certificate Password	`GODOT_MACOS_CODESIGN_CERTIFICATE_PASSWORD`
Options / Codesign / Provisioning Profile	`GODOT_MACOS_CODESIGN_PROVISIONING_PROFILE`
Options / Notarization / API UUID	`GODOT_MACOS_NOTARIZATION_API_UUID`
Options / Notarization / API Key	`GODOT_MACOS_NOTARIZATION_API_KEY`
Options / Notarization / API Key ID	`GODOT_MACOS_NOTARIZATION_API_KEY_ID`
Options / Notarization / Apple ID Name	`GODOT_MACOS_NOTARIZATION_APPLE_ID_NAME`
Options / Notarization / Apple ID Password	`GODOT_MACOS_NOTARIZATION_APPLE_ID_PASSWORD`

Running Godot apps on macOS

See also

This page covers running Godot projects on macOS. If you haven't exported your project yet, read Exporting for macOS first.

By default, macOS will run only applications that are signed and notarized.

Note

When running an app from the Downloads folder or when still in quarantine, Gatekeeper will perform *path randomization* as a security measure. This breaks access to relative paths from the app, which the app relies upon to work. To resolve this issue, move the app to the `/Applications` folder.

In general, macOS apps should avoid relying on relative paths from the application folder.

Depending on the way a macOS app is signed and distributed, the following scenarios are possible:

App is signed, notarized and distributed via App Store

Note

App developers need to join the Apple Developer Program, and configure signing and notarization options during export, then upload the app to the App Store.

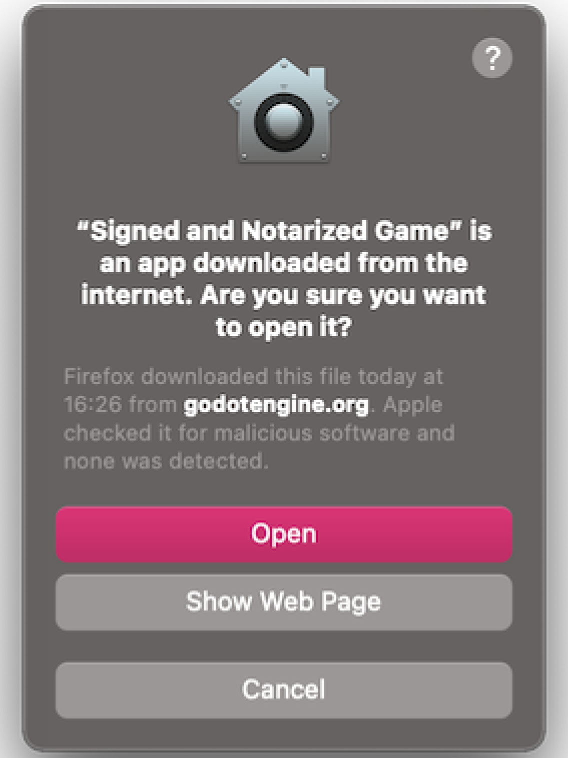
The app should run out of the box, without extra user interaction required.

App is signed, notarized and distributed outside App Store

Note

App developers need to join the Apple Developer Program, and configure signing and notarization options during export, then distribute the app as ".DMG" or ".ZIP" archive.

When you run the app for the first time, the following dialog is displayed:

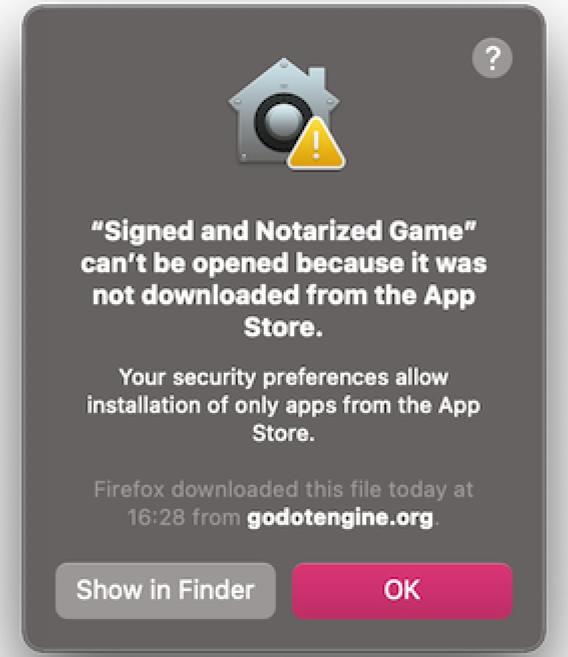


“Signed and Notarized Game” is an app downloaded from the internet. Are you sure you want to open it?

Firefox downloaded this file today at 16:26 from **godotengine.org**. Apple checked it for malicious software and none was detected.

Click `Open` to start the app.

If you see the following warning dialog, your Mac is set up to allow apps only from the App Store.

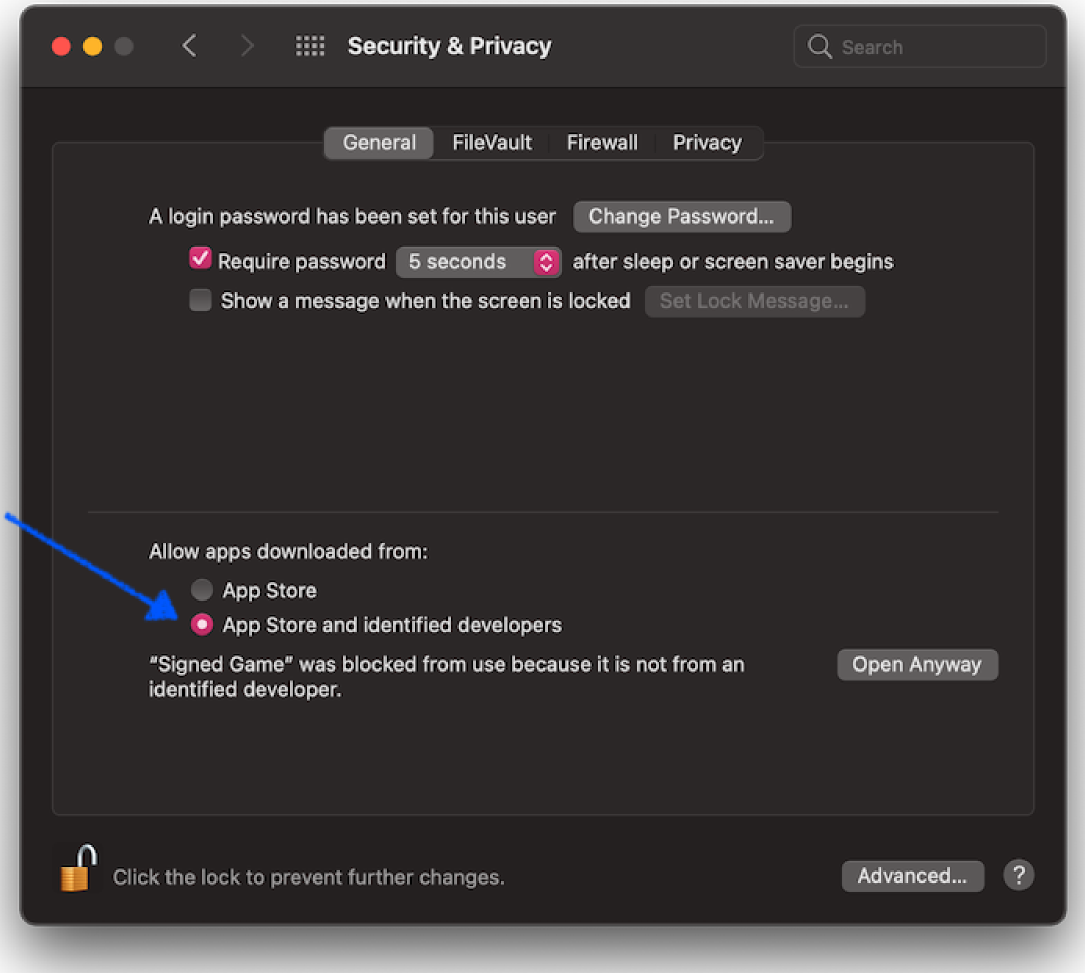


“Signed and Notarized Game” can’t be opened because it was not downloaded from the App Store.

Your security preferences allow installation of only apps from the App Store.

Firefox downloaded this file today at 16:28 from **godotengine.org**.

To allow third-party apps, open `System Preferences`, click `Security & Privacy`, then click `General`, unlock settings, and select `App Store and identified developers`.



Security & Privacy
Search
General
FileVault
Firewall
Privacy
Change Password...
Set Lock Message...
Open Anyway
Advanced...
?

A login password has been set for this user

☑ Require password 5 seconds after sleep or screen saver begins

☐ Show a message when the screen is locked

Allow apps downloaded from:

○ App Store

● App Store and identified developers

“Signed Game” was blocked from use because it is not from an identified developer.

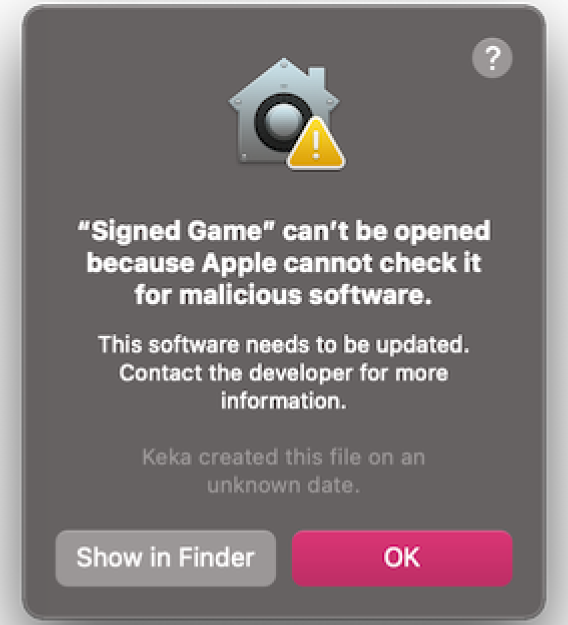
Click the lock to prevent further changes.

App is signed (including ad-hoc signatures) but not notarized

Note

App developer used self-signed certificate or ad-hoc signing (default Godot behavior for exported project).

When you run the app for the first time, the following dialog is displayed:



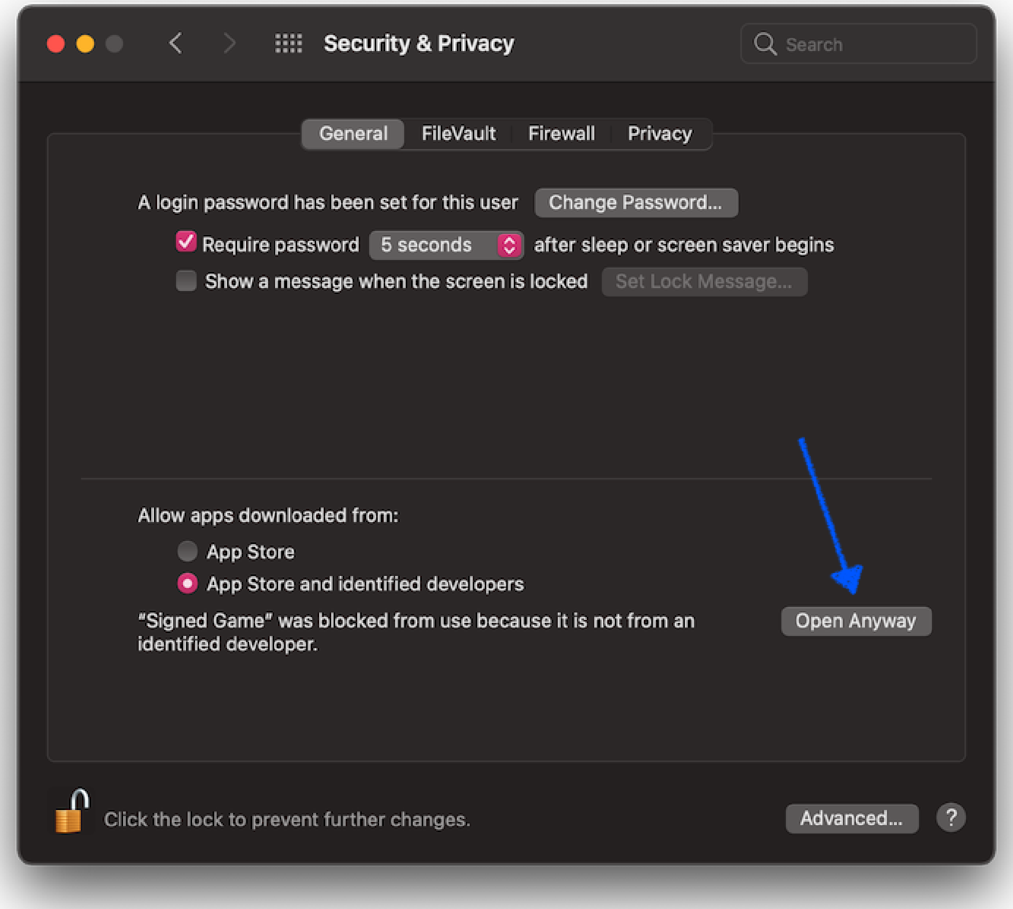
“Signed Game” can’t be opened because Apple cannot check it for malicious software.

This software needs to be updated. Contact the developer for more information.

Keka created this file on an unknown date.

To run this app, you can temporarily override Gatekeeper:

- Either open `System Preferences`, click `Security & Privacy`, then click `General`, and click `Open Anyway`.


Security & Privacy
Search
General
FileVault
Firewall
Privacy
Change Password...
Set Lock Message...
Open Anyway
Advanced...
?

A login password has been set for this user

☑ Require password 5 seconds after sleep or screen saver begins

☐ Show a message when the screen is locked

Allow apps downloaded from:

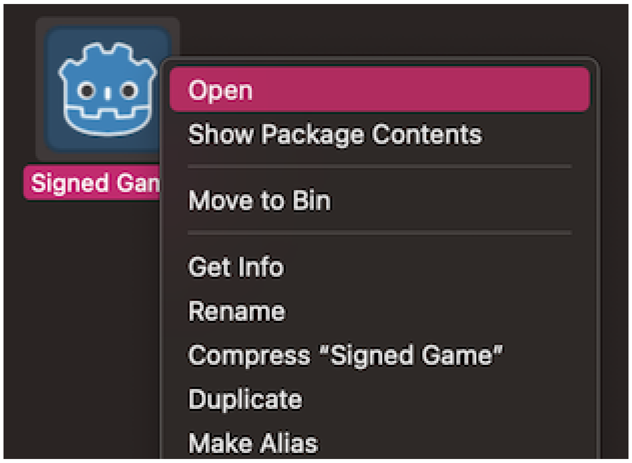
○ App Store

● App Store and identified developers

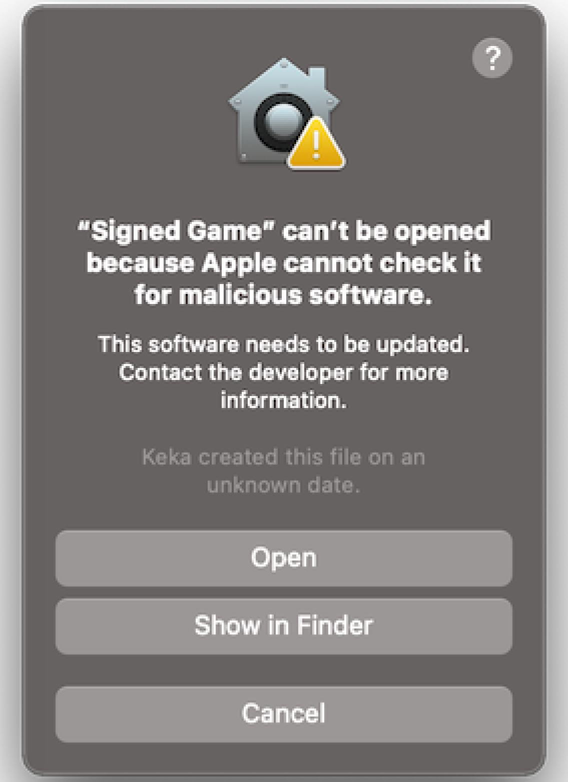
“Signed Game” was blocked from use because it is not from an identified developer.

Click the lock to prevent further changes.

- Or, right-click (Control-click) on the app icon in the Finder window and select `Open` from the menu.


Signed Gan
Open
Show Package Contents
Move to Bin
Get Info
Rename
Compress “Signed Game”
Duplicate
Make Alias

- Then click `Open` in the confirmation dialog.



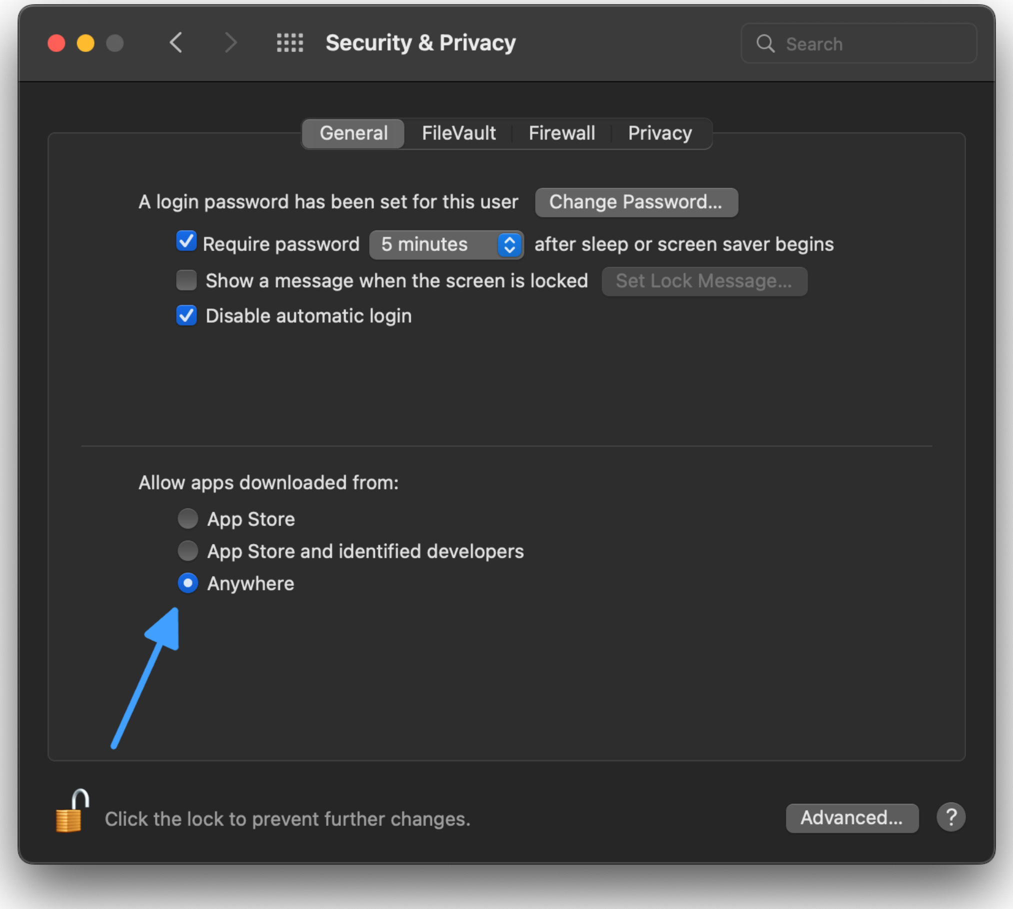
“Signed Game” can’t be opened because Apple cannot check it for malicious software.

This software needs to be updated. Contact the developer for more information.

Keka created this file on an unknown date.

- Enter your password if you're prompted.

Another option is to disable Gatekeeper entirely. Note that this does decrease the security of your computer by allowing you to run any software you want. To do this, run `sudo spctl --master-disable` in the Terminal, enter your password, and then the **Anywhere** option will be available:


Security & Privacy
Search
General
FileVault
Firewall
Privacy
Change Password...
Set Lock Message...
Advanced...
?

A login password has been set for this user

☑ Require password 5 minutes after sleep or screen saver begins

☐ Show a message when the screen is locked

☑ Disable automatic login

Allow apps downloaded from:

○ App Store

○ App Store and identified developers

● Anywhere

Click the lock to prevent further changes.

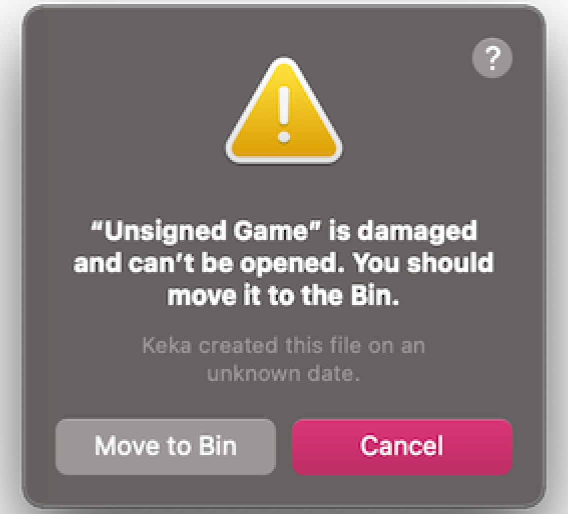
Note that Gatekeeper will re-enable itself when macOS updates.

App is not signed, executable is linker-signed

Note

App is built using official export templates, but it is not signed.

When you run the app for the first time, the following dialog is displayed:



“Unsigned Game” is damaged and can’t be opened. You should move it to the Bin.

Keka created this file on an unknown date.

To run this app, you should remove the quarantine extended file attribute manually:

- Open `Terminal.app` (press `Cmd + Space` and enter `Terminal`).
- Navigate to the folder containing the target application.

  Use the `cd path_to_the_app_folder` command, e.g. `cd ~/Downloads/` if it's in the `Downloads` folder.
- Run the command `xattr -dr com.apple.quarantine "Unsigned Game.app"` (including quotation marks and `.app` extension).

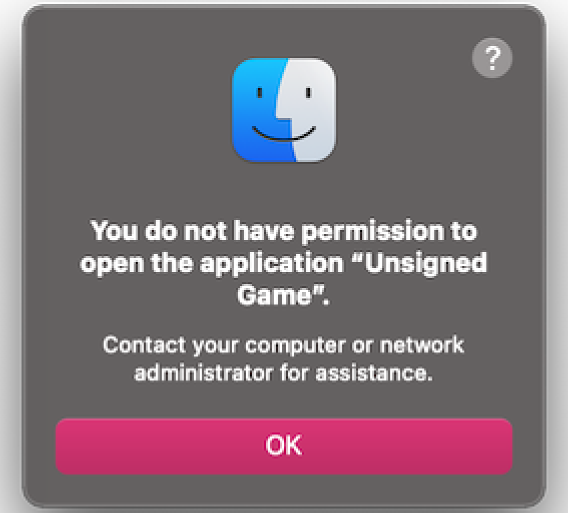
Neither app nor executable is signed (relevant for Apple Silicon

Macs only)

Note

App is built using custom export templates, compiled using OSXCross, and it is not signed at all.

When you run the app for the first time, the following dialog is displayed:



You do not have permission to open the application "Unsigned Game".

Contact your computer or network administrator for assistance.

To run this app, you can ad-hoc sign it yourself:

- Install `Xcode` for the App Store, start it and confirm command line tools installation.
- Open `Terminal.app` (press `Cmd + Space` and enter `Terminal`).
- Navigate to the folder containing the target application.

Use the `cd path_to_the_app_folder` command, e.g. `cd ~/Downloads/` if it's in the `Downloads` folder.

- Run the following commands:

  ```
  xattr -dr com.apple.quarantine "Unsigned Game.app"
  ```
  (including quotation marks and ".app" extension).

  ```
  codesign -s - --force --deep "Unsigned Game.app"
  ```
  (including quotation marks and ".app" extension).

Exporting for Windows

See also

This page describes how to export a Godot project to Windows. If you're looking to compile editor or export template binaries from source instead, read Compiling for Windows.

The simplest way to distribute a game for PC is to copy the executable (`godot.exe`), compress the folder and send it to someone else. However, this is often not desired.

Godot offers a more elegant approach for PC distribution when using the export system. When exporting for Windows, the exporter takes all the project files and creates a `data.pck` file. This file is bundled with a specially optimized binary that is smaller, faster and does not contain the editor and debugger.

Code signing

Godot is capable of automatic code signing on export. To do this you must have the `Windows SDK` (on Windows) or osslsigncode [https://github.com/mtrojnar/osslsigncode] (on any other OS) installed. You will also need a package signing certificate, information on creating one can be found here [https://docs.microsoft.com/en-us/windows/win32/appxpkg/how-to-create-a-package-signing-certificate?redirectedfrom=MSDN].

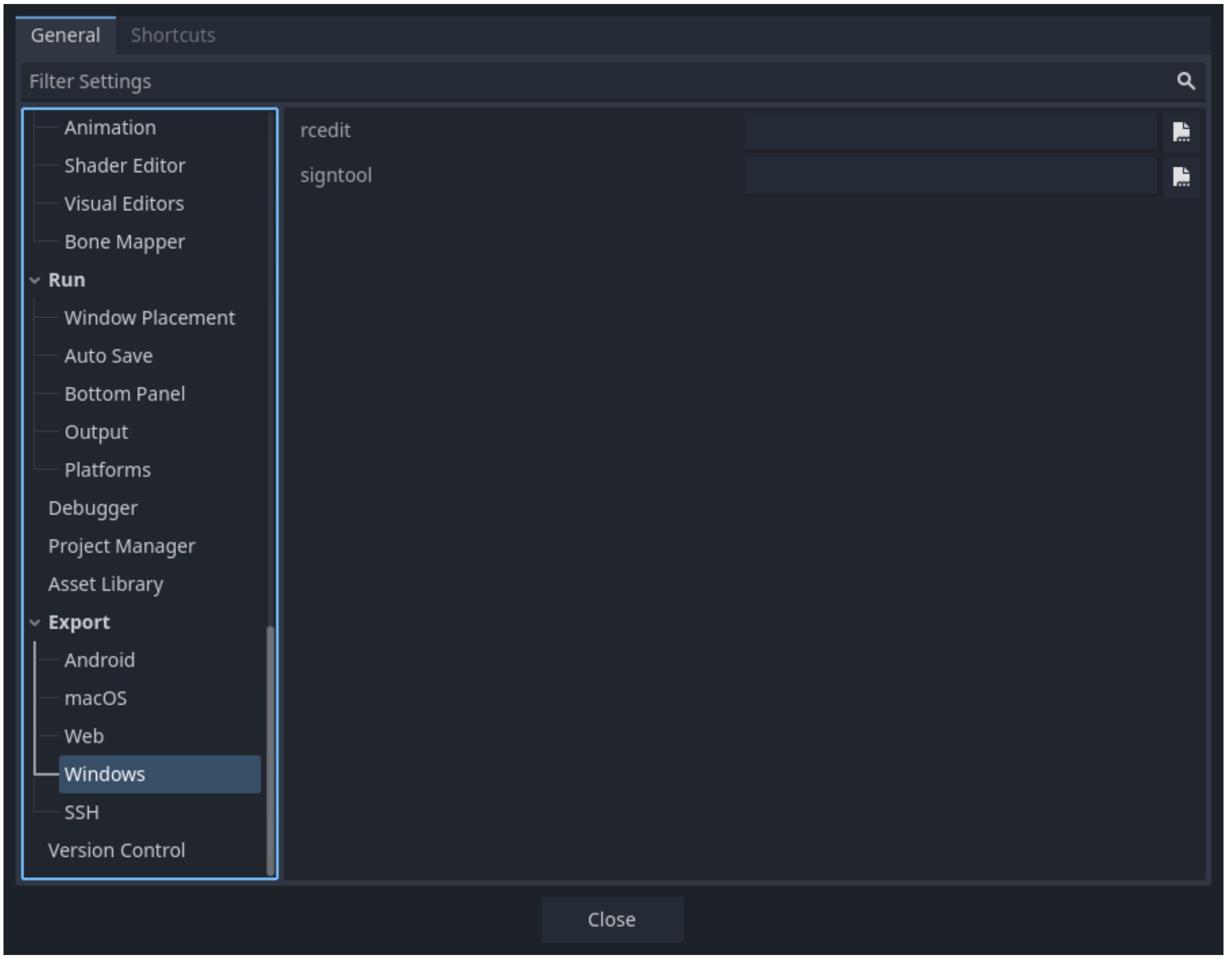
Warning

If you export for Windows with embedded PCK files, you will not be able to sign the program as it will break.

On Windows, PCK embedding is also known to cause false positives in antivirus programs. Therefore, it's recommended to avoid using it unless you're distributing your project via Steam as it bypasses code signing and antivirus checks.

Setup

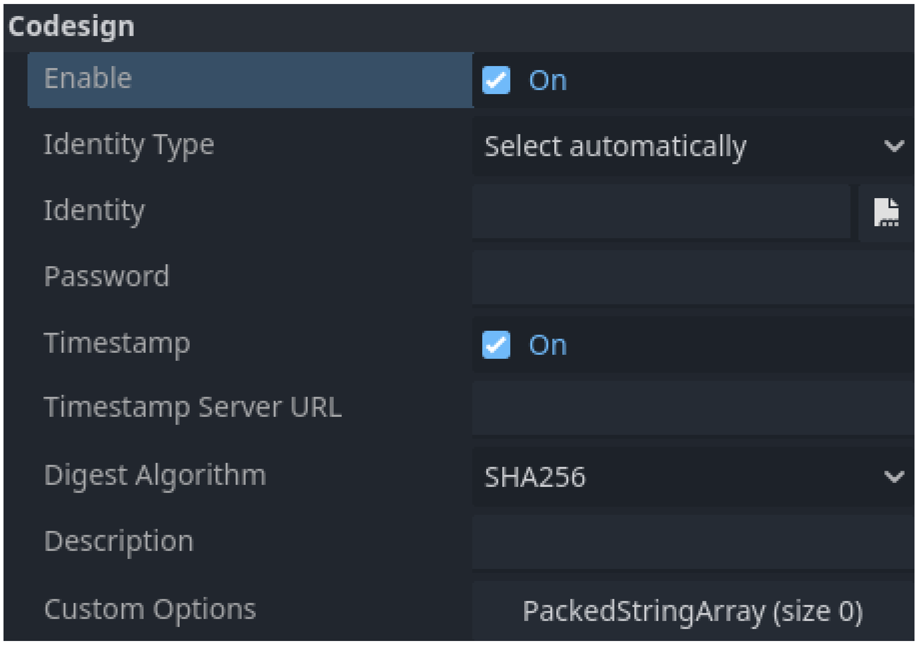
Settings need to be changed in two places. First, in the editor settings, under **Export > Windows**. Click on the folder next to the `Sign Tool` setting, if you're using Windows navigate to and select `SignTool.exe`, if you're on a different OS select `osslsigncode`.


General
Shortcuts
Filter Settings
Animation
Shader Editor
Visual Editors
Bone Mapper
Run
Window Placement
Auto Save
Bottom Panel
Output
Platforms
Debugger
Project Manager
Asset Library
Export
Android
macOS
Web
Windows
SSH
Version Control
Close

rcedit

signtool

The second location is the Windows export preset, which can be found in **Project > Export...**. Add a windows desktop preset if you haven't already. Under options there is a code signing category.


Codesign

Enable	☑ On
Identity Type	Select automatically
Identity	
Password	
Timestamp	☑ On
Timestamp Server URL	
Digest Algorithm	SHA256
Description	
Custom Options	PackedStringArray (size 0)

`Enabled` must be set to true, and `Identity` must be set to the signing certificate. The other settings can be adjusted as needed. Once this is Done Godot will sign your project on export.

Environment variables

You can use the following environment variables to set export options outside of the editor. During the export process, these override the values that you set in the export menu.

Windows export environment variables

Export option	Environment variable
Encryption / Encryption Key	`GODOT_SCRIPT_ENCRYPTION_KEY`
Options / Codesign / Identity Type	`GODOT_WINDOWS_CODESIGN_IDENTITY_TYPE`

Export option	Environment variable
Options / Codesign / Identity	`GODOT_WINDOWS_CODESIGN_IDENTITY`
Options / Codesign / Password	`GODOT_WINDOWS_CODESIGN_PASSWORD`

Changing application icon for Windows

By default, the exported project's icon will be the Godot icon. You will most likely want to change that for your project. There are two types of icons that can be changed on Windows: the file icon and the taskbar icon.

Creating an ICO file

Windows does not use formats such as png or jpg for application icons. Instead, it uses a Windows-only format called ICO. You can create your application icon in any program but you will have to convert it to an ICO file using a program such as GIMP.

This video tutorial [https://www.youtube.com/watch?v=uqV3UfM-n5Y] goes over how to export an ICO file with GIMP.

It is also possible to convert a PNG image to an hiDPI-friendly ICO file using this ImageMagick [https://www.imagemagick.org/] command:

```
magick convert icon.png -define icon:auto-resize=256,128,64,48,32,16 icon.ico
```

Depending on which version of ImageMagick you installed, you might need to leave out the `magick` and run this command instead:

```
convert icon.png -define icon:auto-resize=256,128,64,48,32,16 icon.ico
```

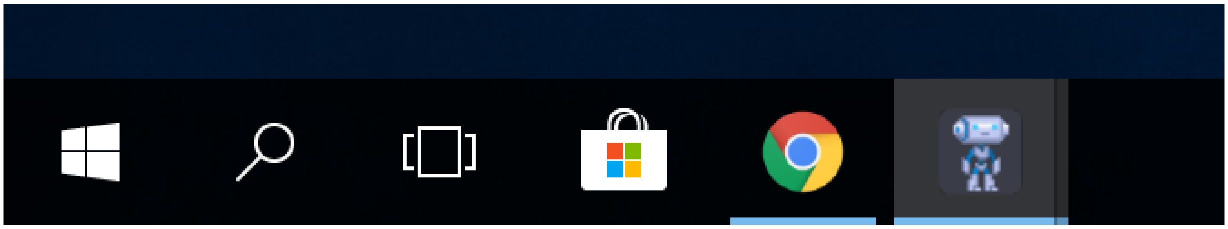
Warning

For the ICO file to effectively replace the default Godot icon, it must contain *all* the sizes included in the default Godot icon: 16×16, 32×32, 48×48, 64×64, 128×128, 256×256. If the ICO file does not contain all the sizes, the default Godot icon will be kept for the sizes that weren't overridden.

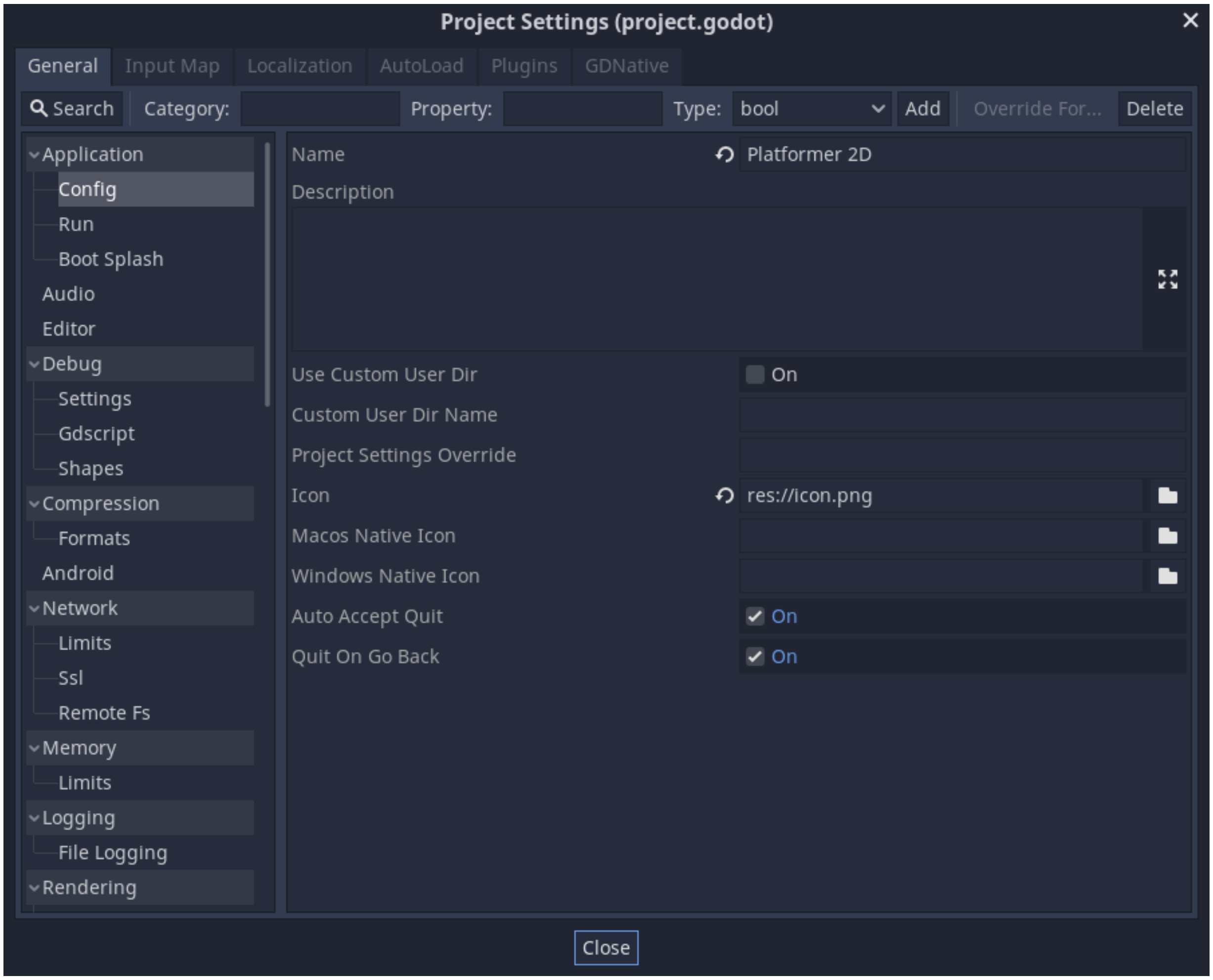
The above ImageMagick command takes this into account.

Changing the taskbar icon

The taskbar icon is the icon that shows up on the taskbar when your project is running.



To change the taskbar icon, go to **Project** → **Project Settings** → **Application** → **Config** → **Windows Native Icon**. Click on the folder icon and select your ICO file.


Project Settings (project.godot)
General
Input Map
Localization
AutoLoad
Plugins
GDNative
Search
Category:
Property:
Type:
bool
Add
Override For...
Delete
Application
Config
Run
Boot Splash
Audio
Editor
Debug
Settings
Gdscript
Shapes
Compression
Formats
Android
Network
Limits
Ssl
Remote Fs
Memory
Limits
Logging
File Logging
Rendering
Close

Name: Platformer 2D
Description
Use Custom User Dir: ☐ On
Custom User Dir Name
Project Settings Override
Icon: res://icon.png
Macos Native Icon
Windows Native Icon
Auto Accept Quit: ☑ On
Quit On Go Back: ☑ On

This setting only changes the icon for your exported game on Windows. To set the icon for macOS, use `Macos Native Icon`. And for any other platform, use the `Icon` setting.

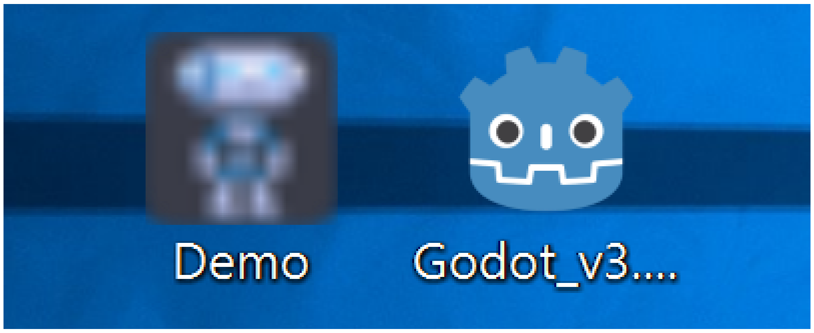
Changing the file icon

In Godot 3.5 and later, you can change the file icon without external tools using godoticon [https://github.com/pkowal1982/godoticon]. Changing the file icon this way should work for executables containing an embedded PCK.

Warning

There are known issues [https://github.com/godotengine/godot/issues/33466] when changing the application icon in executables that embed a PCK file. It's recommended to avoid using rcedit for now if you choose to enable the **Embed Pck** option for your Windows export preset in the Export dialog.

The file icon is the icon of the executable that you click on to start the project.



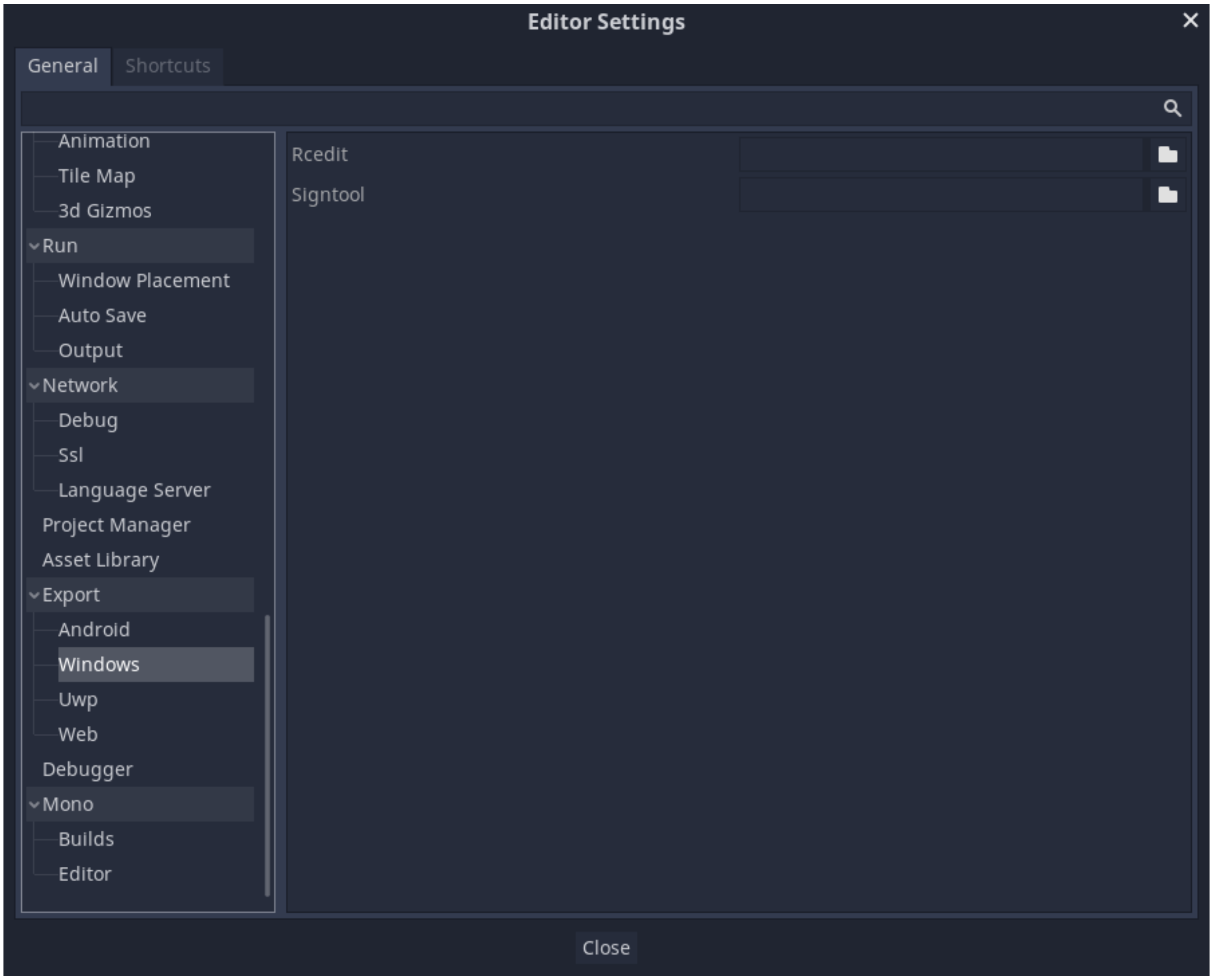
Before selecting it in the export options, you will need to install an extra tool called **rcedit**. You can download it here [https://github.com/electron/rcedit/releases].

After downloading, you need to tell Godot the path to the rcedit executable on your computer. Go to **Editor** → **Editor Settings** → **Export** → **Windows**. Click on the folder icon for the **rcedit** entry. Navigate to and select the rcedit executable.

Note

Linux and macOS users will also need to install WINE [https://www.winehq.org/] to use rcedit.

On macOS 10.15 (Catalina) or later, make sure to use a 64-bit version of both WINE and rcedit since 32-bit applications aren't supported anymore.

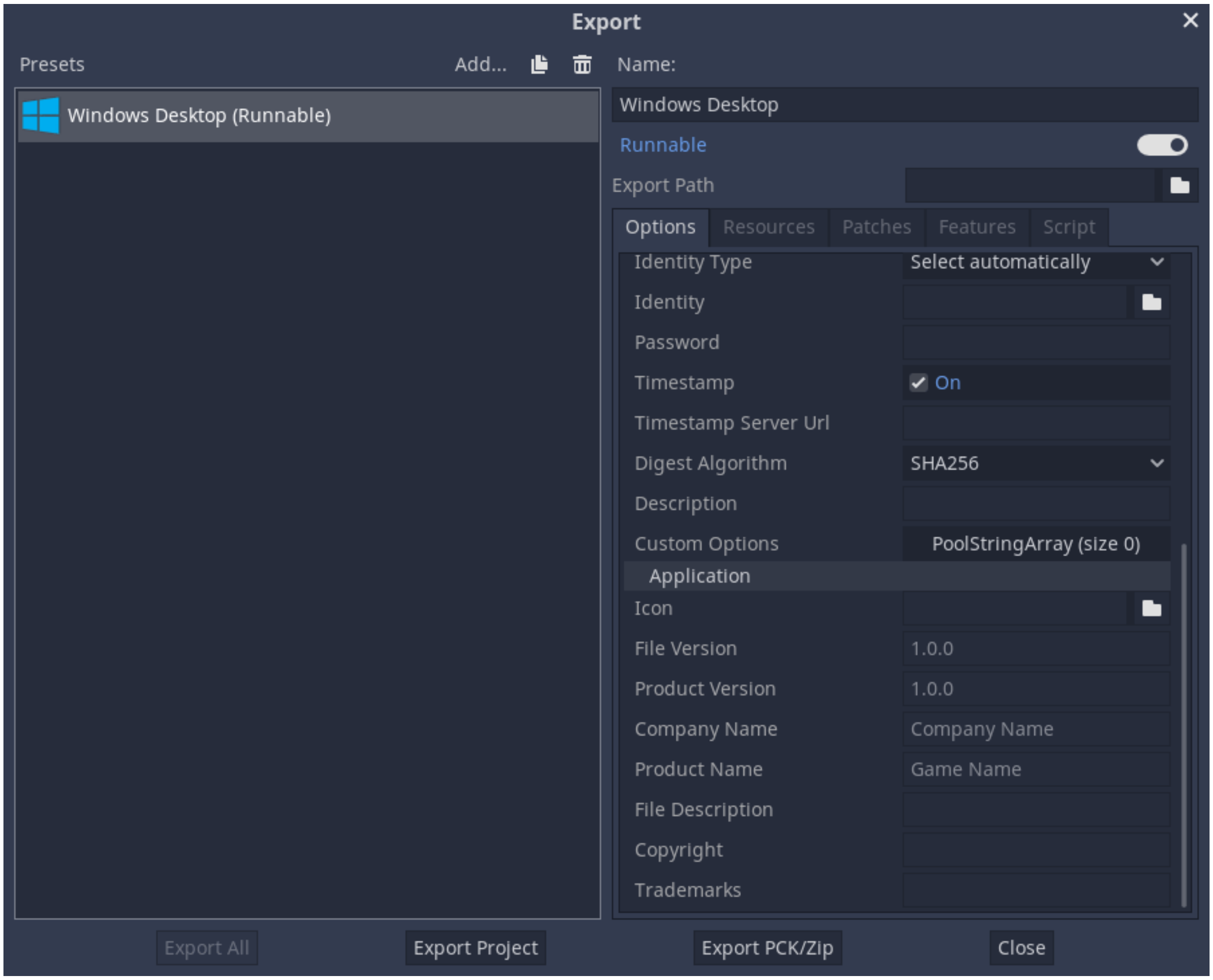

Editor Settings
General
Shortcuts
Animation
Tile Map
3d Gizmos
Run
Window Placement
Auto Save
Output
Network
Debug
Ssl
Language Server
Project Manager
Asset Library
Export
Android
Windows
Uwp
Web
Debugger
Mono
Builds
Editor
Rcedit
Signtool
Close

You should now have everything in place to change the file icon. To do that, you will need to specify the icon when exporting. Go to **Project** → **Export**. Assuming you have already created a Windows Desktop preset, select your icon in ICO format in the **Application** → **Icon** field.

Warning

All the supplied metadata must be valid. Most importantly, the version identifiers **must** take the form `major.minor.patch.revision` where every component is required and is a positive integer. For instance, `1.2.0.0` is valid but `1.2.0` is not.

If you provide invalid metadata in the export preset, the application icon won't change as rcedit will fail to change the executable's metadata.


Export
Presets
Add...
Options
Resources
Patches
Features
Script
Export All
Export Project
Export PCK/Zip
Close

Windows Desktop (Runnable)

Name: Windows Desktop

Runnable: On

Export Path:

Identity Type: Select automatically

Identity:

Password:

Timestamp: ☑ On

Timestamp Server Url:

Digest Algorithm: SHA256

Description:

Custom Options: PoolStringArray (size 0)

Application

Icon:

File Version: 1.0.0

Product Version: 1.0.0

Company Name: Company Name

Product Name: Game Name

File Description:

Copyright:

Trademarks:

Note

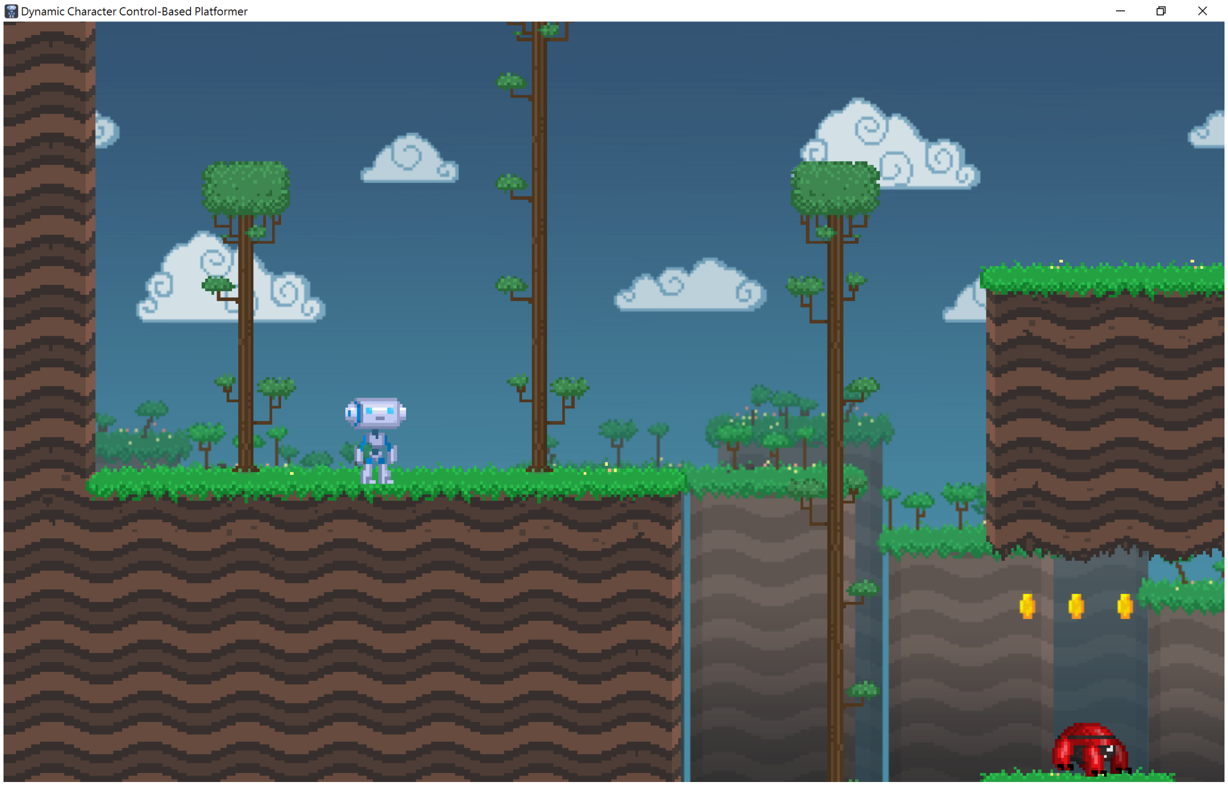
If rcedit fails to change the icon, you can instead [compile your own Windows export templates]() with the icon changed. To do so, replace [platform/windows/godot.ico]()

[https://github.com/godotengine/godot/blob/master/platform/windows/godot.ico] with your own ICO file *before* compiling export templates.

Once this is done, you can specify your export templates as custom export templates in your project's Windows export preset.

Testing the result

You can now export the project. If it worked correctly, you should see this:



Note

If your icon isn't showing up properly, on Windows 10, try clearing the icon cache. To do so, open the **Run** dialog and enter `ie4uinit.exe -ClearIconCache` or `ie4uinit.exe -show`.

Exporting for iOS

See also

This page describes how to export a Godot project to iOS. If you're looking to compile export template binaries from source instead, read Compiling for iOS.

These are the steps to load a Godot project in Xcode. This allows you to build and deploy to an iOS device, build a release for the App Store, and do everything else you can normally do with Xcode.

Attention

Projects written in C# can be exported to iOS as of Godot 4.2, but support is experimental and some limitations apply.

Requirements

- You must export for iOS from a computer running macOS with Xcode installed.
- Download the Godot export templates. Use the Godot menu: Editor > Manage Export Templates

Export a Godot project to Xcode

In the Godot editor, open the **Export** window from the **Project** menu. When the Export window opens, click **Add..** and select **iOS**.

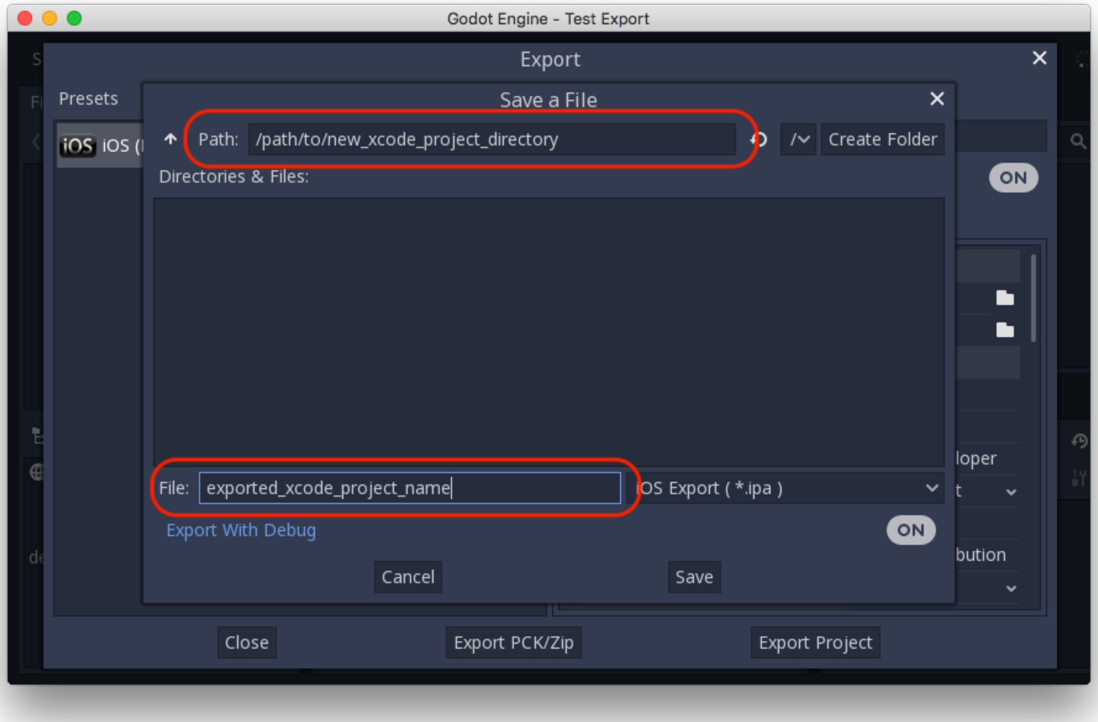
The **App Store Team ID** and (Bundle) **Identifier** options in the **Application** category are required. Leaving them blank will cause the exporter to throw an error.

Note

If you encounter an error during export similar to `JSON text did not start with array or object and option to allow fragments not set` then it might be due to a malformated **App Store Team ID**! The exporter expects a (10 characters long) code like `ABCDE12XYZ` and not, e.g., your name as Xcode likes to display in the *Signing & Capabilities* tab. You can find the code over at developer.apple.com [https://developer.apple.com/account/resources/certificates/list] next to your name in the top right corner.

After you click **Export Project**, there are still two important options left:

- **Path** is an empty folder that will contain the exported Xcode project files.
- **File** will be the name of the Xcode project and several project specific files and directories.



Godot Engine - Test Export
Export
Presets
iOS
Save a File
Create Folder
Directories & Files:
iOS Export (*.ipa)
Export With Debug
ON
Cancel
Save
Close
Export PCK/Zip
Export Project

Path: /path/to/new_xcode_project_directory

File: exported_xcode_project_name

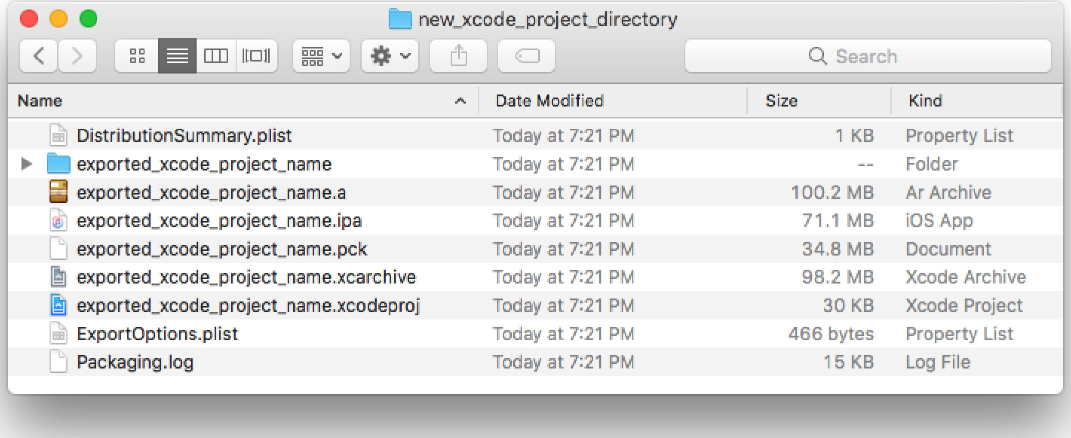
Note

This tutorial uses **exported_xcode_project_name**, but you will use your project's name. When you see **exported_xcode_project_name** in the following steps, replace it with the name you used instead.

Note

Avoid using spaces when you choose your **exported_xcode_project_name** as this can lead to corruption in your XCode project file.

When the export completes, the output folder should look like this:



Name	Date Modified	Size	Kind
DistributionSummary.plist	Today at 7:21 PM	1 KB	Property List
exported_xcode_project_name	Today at 7:21 PM	--	Folder
exported_xcode_project_name.a	Today at 7:21 PM	100.2 MB	Ar Archive
exported_xcode_project_name.ipa	Today at 7:21 PM	71.1 MB	iOS App
exported_xcode_project_name.pck	Today at 7:21 PM	34.8 MB	Document
exported_xcode_project_name.xcarchive	Today at 7:21 PM	98.2 MB	Xcode Archive
exported_xcode_project_name.xcodeproj	Today at 7:21 PM	30 KB	Xcode Project
ExportOptions.plist	Today at 7:21 PM	466 bytes	Property List
Packaging.log	Today at 7:21 PM	15 KB	Log File

Opening **exported_xcode_project_name.xcodeproj** lets you build and deploy like any other iOS app.

Active development considerations

The above method creates an exported project that you can build for release, but you have to re-export every time you make a change in Godot.

While developing, you can speed this process up by linking your Godot project files directly into your app.

In the following example:

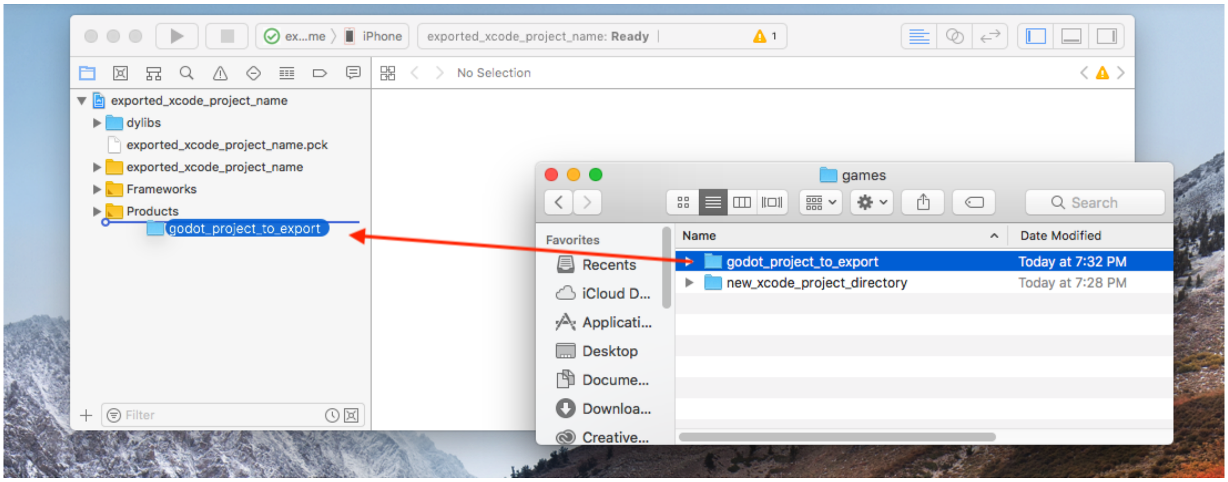
- **exported_xcode_project_name** is the name of the exported iOS application (as above).
- **godot_project_to_export** is the name of the Godot project.

Note

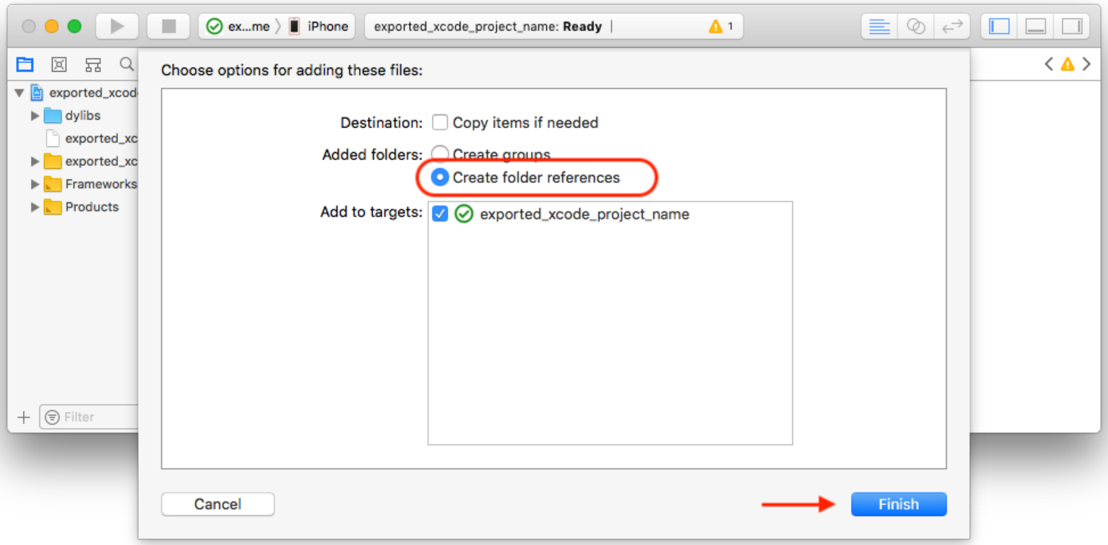
godot_project_to_export must not be the same as **exported_xcode_project_name** to prevent signing issues in Xcode.

Steps to link a Godot project folder to Xcode

1. Start from an exported iOS project (follow the steps above).
2. In Finder, drag the Godot project folder into the Xcode file browser.


ex...me
iPhone
exported_xcode_project_name: Ready
No Selection
exported_xcode_project_name
dylibs
exported_xcode_project_name.pck
exported_xcode_project_name
Frameworks
Products
godot_project_to_export
Filter
games
Search
Favorites
Recents
iCloud D...
Applicati...
Desktop
Docume...
Downloa...
Creative...
Name
Date Modified
godot_project_to_export
Today at 7:32 PM
new_xcode_project_directory
Today at 7:28 PM

3. In the dialog, make sure **Create folder references** is selected. This means you will be able to continue to edit your Godot project in its current location.


ex...me 〉 iPhone
exported_xcode_project_name: **Ready**
exported_xcod
dylibs
exported_xc
exported_xc
Frameworks
Products
Filter
Cancel
Finish

Choose options for adding these files:

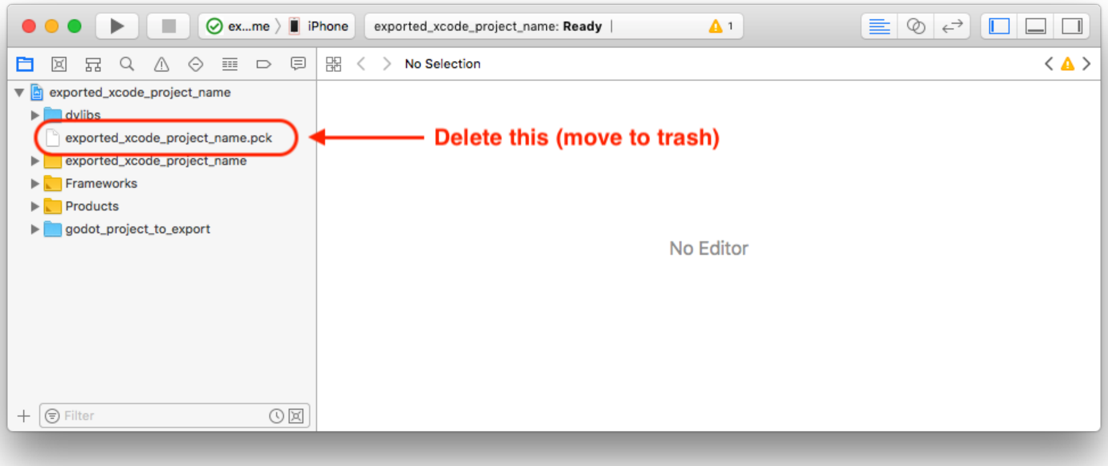
Destination: ☐ Copy items if needed

Added folders: ○ Create groups

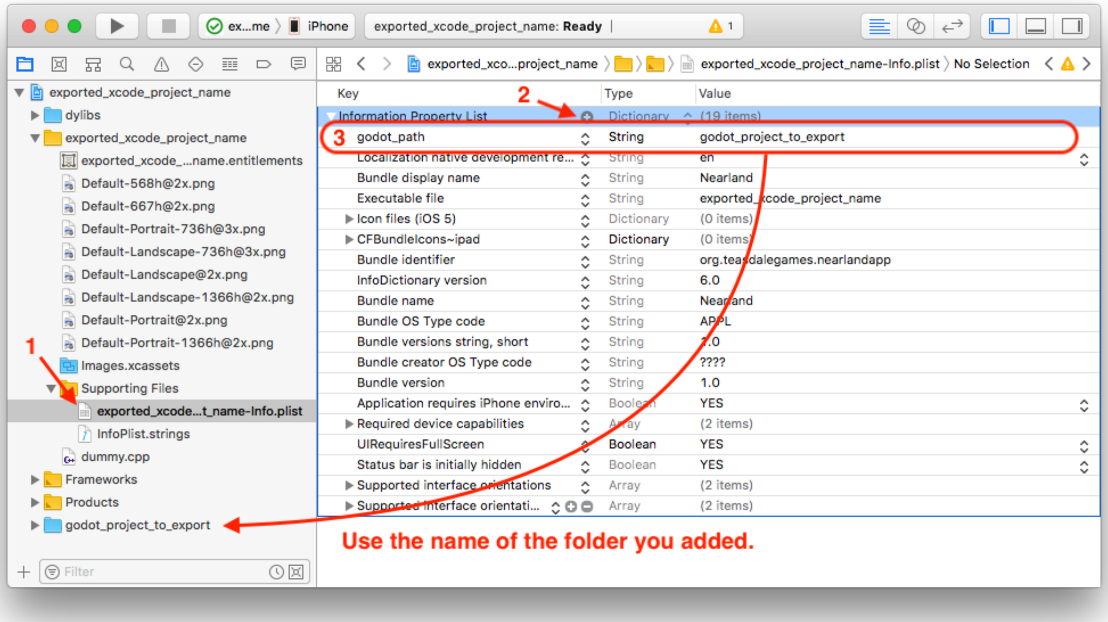
● Create folder references

Add to targets: ☑ exported_xcode_project_name

4. See the **godot_project_to_export** folder in the Xcode file browser.
5. Delete **exported_xcode_project_name.pck** from the Xcode project.


ex...me 〉 iPhone
exported_xcode_project_name: **Ready**
No Selection
exported_xcode_project_name
dylibs
exported_xcode_project_name.pck
exported_xcode_project_name
Frameworks
Products
godot_project_to_export
Delete this (move to trash)
No Editor
Filter

6. Open **exported_xcode_project_name-Info.plist** and add a string property named **godot_path** (this is the real key name) with a value **godot_project_to_export** (this is the name of your project)


ex...me
iPhone
exported_xcode_project_name: Ready
exported_xco...project_name
exported_xcode_project_name-Info.plist
No Selection
exported_xcode_project_name
dylibs
exported_xcode_project_name
exported_xcode_...name.entitlements
Default-568h@2x.png
Default-667h@2x.png
Default-Portrait-736h@3x.png
Default-Landscape-736h@3x.png
Default-Landscape@2x.png
Default-Landscape-1366h@2x.png
Default-Portrait@2x.png
Default-Portrait-1366h@2x.png
Images.xcassets
Supporting Files
exported_xcode...t_name-Info.plist
InfoPlist.strings
dummy.cpp
Frameworks
Products
godot_project_to_export
Filter
1
2
3

Key	Type	Value
Information Property List	Dictionary	(19 items)
godot_path	String	godot_project_to_export
Localization native development re...	String	en
Bundle display name	String	Nearland
Executable file	String	exported_xcode_project_name
Icon files (iOS 5)	Dictionary	(0 items)
CFBundleIcons~ipad	Dictionary	(0 items)
Bundle identifier	String	org.teasdalegames.nearlandapp
InfoDictionary version	String	6.0
Bundle name	String	Nearland
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone enviro...	Boolean	YES
Required device capabilities	Array	(2 items)
UIRequiresFullScreen	Boolean	YES
Status bar is initially hidden	Boolean	YES
Supported interface orientations	Array	(2 items)
Supported interface orientati...	Array	(2 items)

Use the name of the folder you added.

That's it! You can now edit your project in the Godot editor and build it in Xcode when you want to run it on a device.

Plugins for iOS

Special iOS plugins can be used in Godot. Check out the Plugins for iOS page.

Environment variables

You can use the following environment variables to set export options outside of the editor. During the export process, these override the values that you set in the export menu.

iOS export environment variables

Export option	Environment variable

Export option	**Environment variable**
Encryption / Encryption Key	`GODOT_SCRIPT_ENCRYPTION_KEY`
Options / Application / Provisioning Profile UUID Debug	`GODOT_IOS_PROVISIONING_PROFILE_UUID_DEBUG`
Options / Application / Provisioning Profile UUID Release	`GODOT_IOS_PROVISIONING_PROFILE_UUID_RELEASE`

Troubleshooting

xcode-select points at wrong SDK location

xcode-select is a tool that comes with Xcode and among other things points at iOS SDKs on your Mac. If you have Xcode installed, opened it, agreed to the license agreement, and installed the command line tools, xcode-select should point at the right location for the iPhone SDK. If it somehow doesn't, Godot will fail exporting to iOS with an error that may look like this:

```
MSB3073: The command ""clang" <LOTS OF PATHS AND COMMAND LINE
ARGUMENTS HERE>
"/Library/Developer/CommandLineTools/Platforms/iPhoneOS.platfor
m/Developer/SDKs/iPhoneOS.sdk"" exited with code 1.
```

In this case, Godot is trying to find the `Platforms` folder containing the iPhone SDK inside the `/Library/Developer/CommandLineTools/` folder, but the `Platforms` folder with the iPhone SDK is actually located under `/Applications/Xcode.app/Contents/Developer`. To verify this, you can open up Terminal and run the following command to see what xcode-select points at:

```
xcode-select -p
```

To fix xcode-select pointing at a wrong location, enter this command in Terminal:

```
sudo xcode-select -switch /Applications/Xcode.app
```

After running this command, Godot should be able to successfully export to iOS.

Exporting for Android

See also

This page describes how to export a Godot project to Android. If you're looking to compile export template binaries from source instead, read Compiling for Android.

Exporting for Android has fewer requirements than compiling Godot for Android. The following steps detail what is needed to set up the Android SDK and the engine.

Attention

Projects written in C# can be exported to Android as of Godot 4.2, but support is experimental and some limitations apply.

Install OpenJDK 17

Download and install OpenJDK 17 [https://adoptium.net/temurin/releases/?variant=openjdk17].

Download the Android SDK

Download and install the Android SDK.

- You can install the Android SDK using Android Studio Iguana (version 2023.2.1) or later [https://developer.android.com/studio/].

  - Run it once to complete the SDK setup using these instructions [https://developer.android.com/studio/intro/update#sdk-manager].
  - Ensure that the required packages [https://developer.android.com/studio/intro/update#required] are installed as well.
    - Android SDK Platform-Tools version 34.0.0 or later
    - Android SDK Build-Tools version 34.0.0
    - Android SDK Platform 34
    - Android SDK Command-line Tools (latest)
  - Ensure that the NDK and CMake are installed and configured [https://developer.android.com/studio/projects/install-ndk].
    - CMake version 3.10.2.4988404
    - NDK version r23c (23.2.8568313)
- Alternatively, you can install the Android SDK with the *sdkmanager* command line tool.
  - Install the command line tools package using these instructions [https://developer.android.com/tools/sdkmanager].
  - Once the command line tools are installed, run the following *sdkmanager* command to complete the setup process:

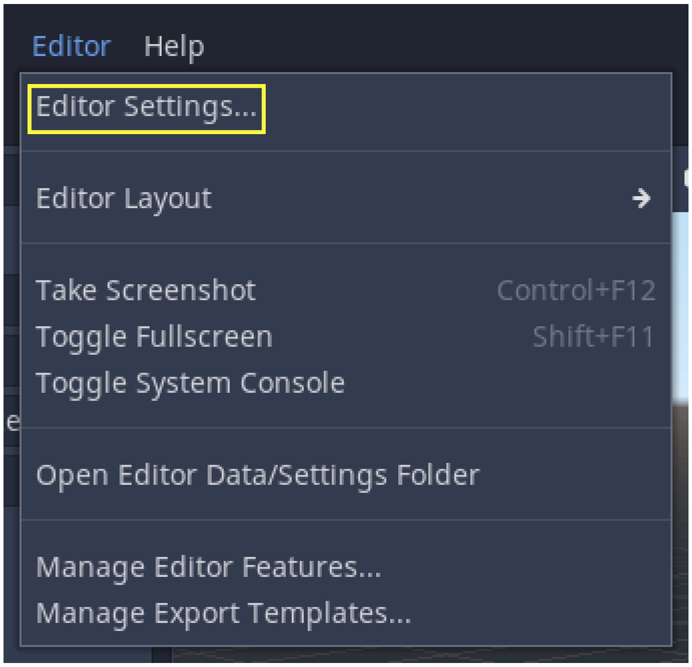
```
sdkmanager --sdk_root=<android_sdk_path> "platform-tools"
"build-tools;34.0.0" "platforms;android-34" "cmdline-
tools;latest" "cmake;3.10.2.4988404" "ndk;23.2.8568313"
```

Note

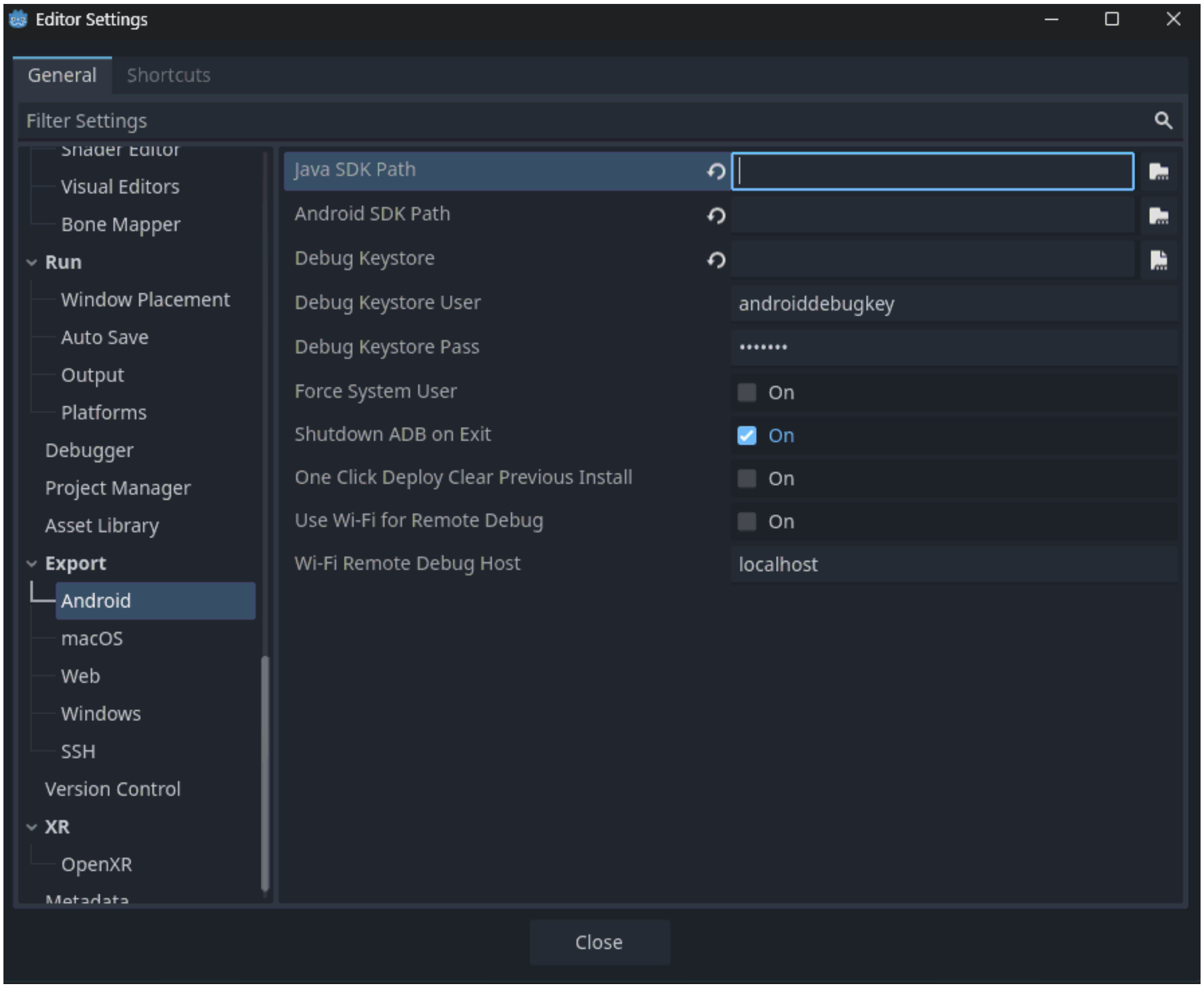
If you are using Linux, **do not use an Android SDK provided by your distribution's repositories as it will often be outdated**.

Setting it up in Godot

Enter the Editor Settings screen. This screen contains the editor settings for the user account in the computer (it's independent of the project).


Editor
Help
Editor Settings...
Editor Layout
Take Screenshot Control+F12
Toggle Fullscreen Shift+F11
Toggle System Console
Open Editor Data/Settings Folder
Manage Editor Features...
Manage Export Templates...

Scroll down to the section where the Android settings are located:


Editor Settings
General
Shortcuts
Filter Settings
Shader Editor
Visual Editors
Bone Mapper
Run
Window Placement
Auto Save
Output
Platforms
Debugger
Project Manager
Asset Library
Export
Android
macOS
Web
Windows
SSH
Version Control
XR
OpenXR
Metadata
Close

Java SDK Path	
Android SDK Path	
Debug Keystore	
Debug Keystore User	androiddebugkey
Debug Keystore Pass	•••••••
Force System User	☐ On
Shutdown ADB on Exit	☑ On
One Click Deploy Clear Previous Install	☐ On
Use Wi-Fi for Remote Debug	☐ On
Wi-Fi Remote Debug Host	localhost

In that screen, 2 paths need to be set:

- `Java SDK Path` should be the location where OpenJDK 17 was installed.
- `Android Sdk Path` should be the location where the Android SDK was installed. - For example `%LOCALAPPDATA%\Android\Sdk\` on Windows or `/Users/$USER/Library/Android/sdk/` on macOS.

Once that is configured, everything is ready to export to Android!

Note

If you get an error saying *"Could not install to device."*, make sure you do not have an application with the same Android package name already installed on the device (but signed with a different key).

If you have an application with the same Android package name but a different signing key already installed on the device, you **must** remove the application in question from the Android device before exporting to Android again.

Providing launcher icons

Launcher icons are used by Android launcher apps to represent your application to users. Godot only requires high-resolution icons (for `xxxhdpi` density screens) and will automatically generate lower-resolution variants.

There are two types of icons required by Godot:

- **Main Icon:** The "classic" icon. This will be used on all Android versions up to Android 8 (Oreo), exclusive. Must be at least 192×192 px.
- **Adaptive Icons:** Starting from Android 8 (inclusive), Adaptive Icons [https://developer.android.com/guide/practices/ui_guidelines/icon_design_adaptive] were introduced. Applications will need to include separate background and foreground icons to have a native look. The user's launcher application will control the icon's animation and masking. Must be at least 432×432 px.

See also

It's important to adhere to some rules when designing adaptive icons. Google Design has provided a nice article [https://medium.com/google-design/designing-adaptive-icons-515af294c783] that helps to understand those rules and some of the capabilities of adaptive icons.

Caution

The most important adaptive icon design rule is to have your icon critical elements inside the safe zone: a centered circle with a diameter of 66dp (264 pixels on `xxxhdpi`) to avoid being clipped by the launcher.

If you don't provide some of the requested icons, Godot will replace them using a fallback chain, trying the next in line when the current one fails:

- **Main Icon:** Provided main icon -> Project icon -> Default Godot main icon.
- **Adaptive Icon Foreground:** Provided foreground icon -> Provided main icon -> Project icon -> Default Godot foreground icon.
- **Adaptive Icon Background:** Provided background icon -> Default Godot background icon.

It's highly recommended to provide all the requested icons with their specified resolutions. This way, your application will look great on all Android devices and versions.

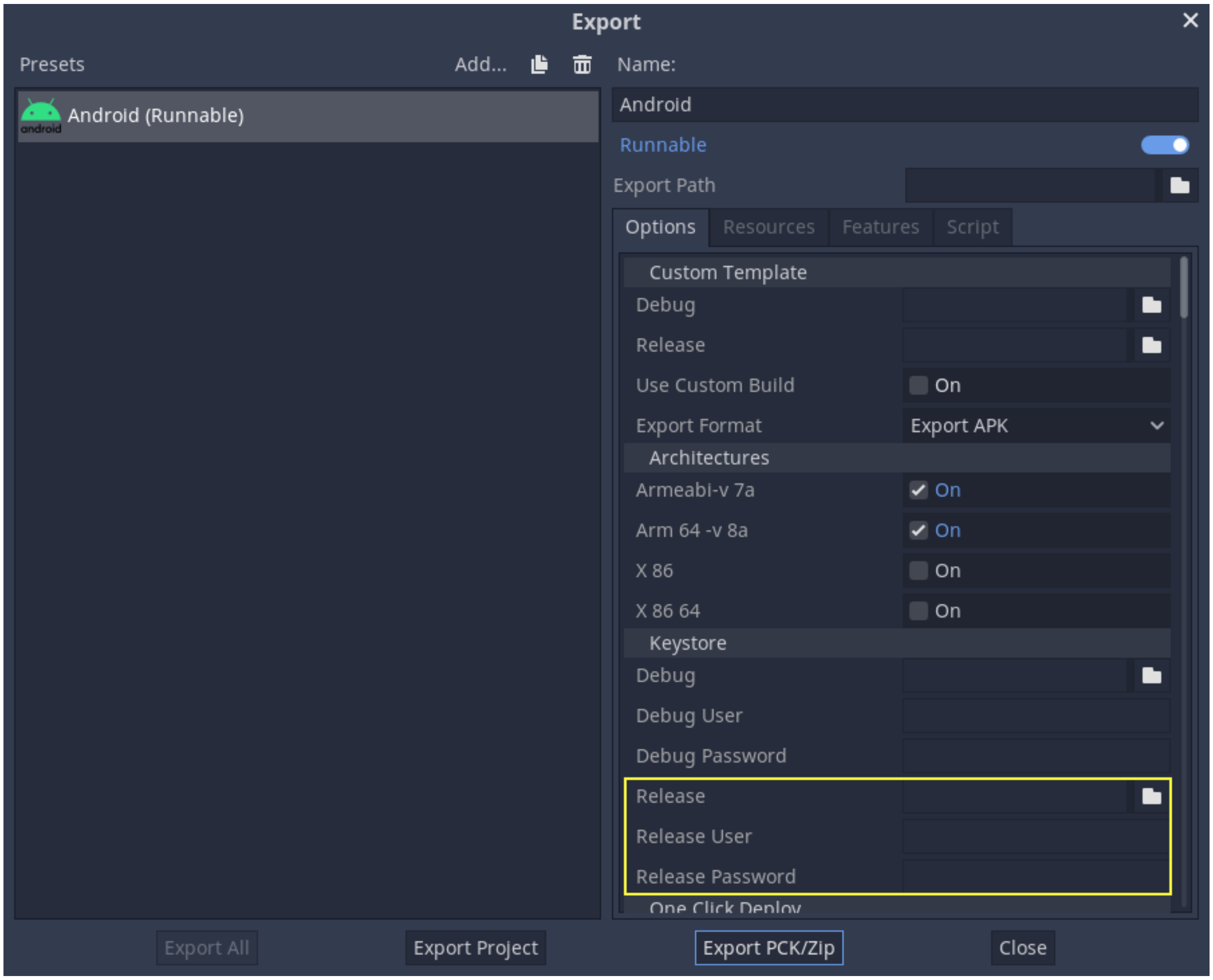
Exporting for Google Play Store

Uploading an APK to Google's Play Store requires you to sign using a non-debug keystore file; such file can be generated like this:

```
keytool -v -genkey -keystore mygame.keystore -alias mygame -keyalg RSA -validity 10000
```

This keystore and key are used to verify your developer identity, remember the password and keep it in a safe place! It is suggested to use only upper and lowercase letters and numbers. Special characters may cause errors. Use Google's Android Developer guides to learn more about APK signing [https://developer.android.com/studio/publish/app-signing].

Now fill in the following forms in your Android Export Presets:


Export
Presets
Add...
Android (Runnable)
Options
Resources
Features
Script
Export All
Export Project
Export PCK/Zip
Close

Name: Android

Runnable ●

Export Path

Custom Template

Debug

Release

Use Custom Build ☐ On

Export Format: Export APK

Architectures

Armeabi-v 7a ☑ On

Arm 64 -v 8a ☑ On

X 86 ☐ On

X 86 64 ☐ On

Keystore

Debug

Debug User

Debug Password

Release

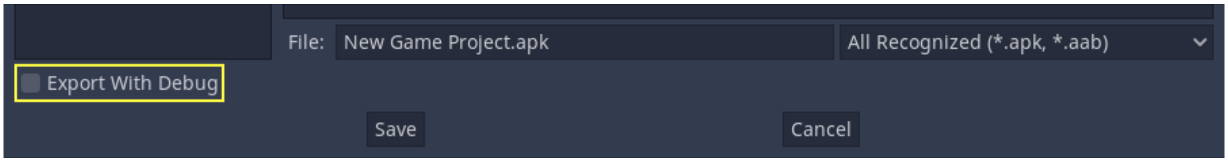
Release User

Release Password

One Click Deploy

- **Release:** Enter the path to the keystore file you just generated.
- **Release User:** Replace with the key alias.
- **Release Password:** Key password. Note that the keystore password and the key password currently have to be the same.

Don't forget to uncheck the **Export With Debug** checkbox while exporting.



Optimizing the APK size

By default, the APK will contain native libraries for both ARMv7 and ARMv8 architectures. This increases its size significantly. To create a smaller APK, uncheck either **Armeabi-v 7a** or **Arm 64 -v 8a** in your project's Android export preset. This will create an APK that only contains a library for a single architecture. Note that applications targeting ARMv7 can also run on ARMv8 devices, but the opposite is not true.

Since August 2019, Google Play requires all applications to be available in 64-bit form. This means you cannot upload an APK that contains *just* an ARMv7 library. To solve this, you can upload several APKs to Google Play using its Multiple APK support [https://developer.android.com/google/play/publishing/multiple-apks]. Each APK should target a single architecture; creating an APK for ARMv7 and ARMv8 is usually sufficient to cover most devices in use today.

You can optimize the size further by compiling an Android export template with only the features you need. See Optimizing a build for size for more information.

Environment variables

You can use the following environment variables to set export options outside of the editor. During the export process, these override the values that you set in the export menu.

Android export environment variables

Export option	Environment variable
Encryption / Encryption Key	`GODOT_SCRIPT_ENCRYPTION_KEY`
Options / Keystore / Debug	`GODOT_ANDROID_KEYSTORE_DEBUG_PATH`
Options / Keystore / Debug User	`GODOT_ANDROID_KEYSTORE_DEBUG_USER`
Options / Keystore / Debug Password	`GODOT_ANDROID_KEYSTORE_DEBUG_PASSWORD`
Options / Keystore / Release	`GODOT_ANDROID_KEYSTORE_RELEASE_PATH`
Options / Keystore / Release User	`GODOT_ANDROID_KEYSTORE_RELEASE_USER`
Options / Keystore / Release Password	`GODOT_ANDROID_KEYSTORE_RELEASE_PASSWORD`

Gradle builds for Android

Godot provides the option to build using the gradle buildsystem. Instead of using the already pre-built template that ships with Godot, an Android Java project gets installed into your project folder. Godot will then build it and use it as an export template every time you export the project.

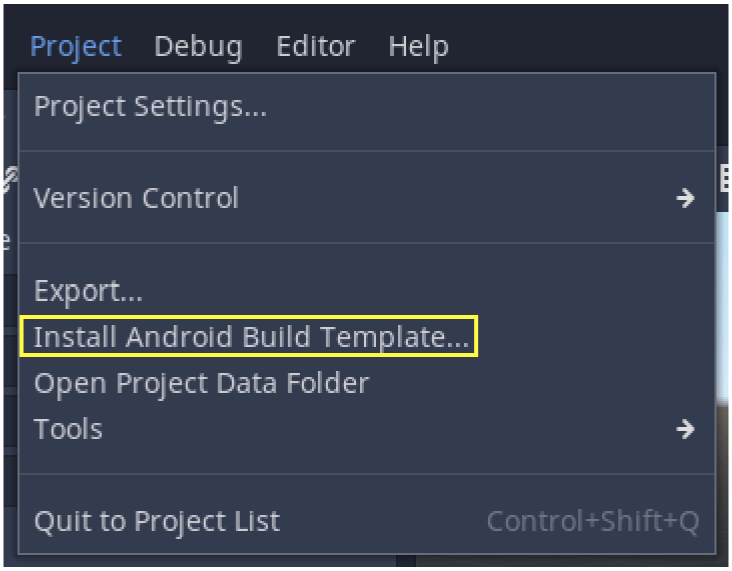
There are some reasons why you may want to do this:

- Modify the project before it's built.
- Add external SDKs that build with your project.

Configuring the gradle build is a fairly straightforward process. But first you need to follow the steps in exporting for android up to **Setting it up in Godot**. After doing that, follow the steps below.

Set up the gradle build environment

Go to the Project menu, and install the *Gradle Build* template:

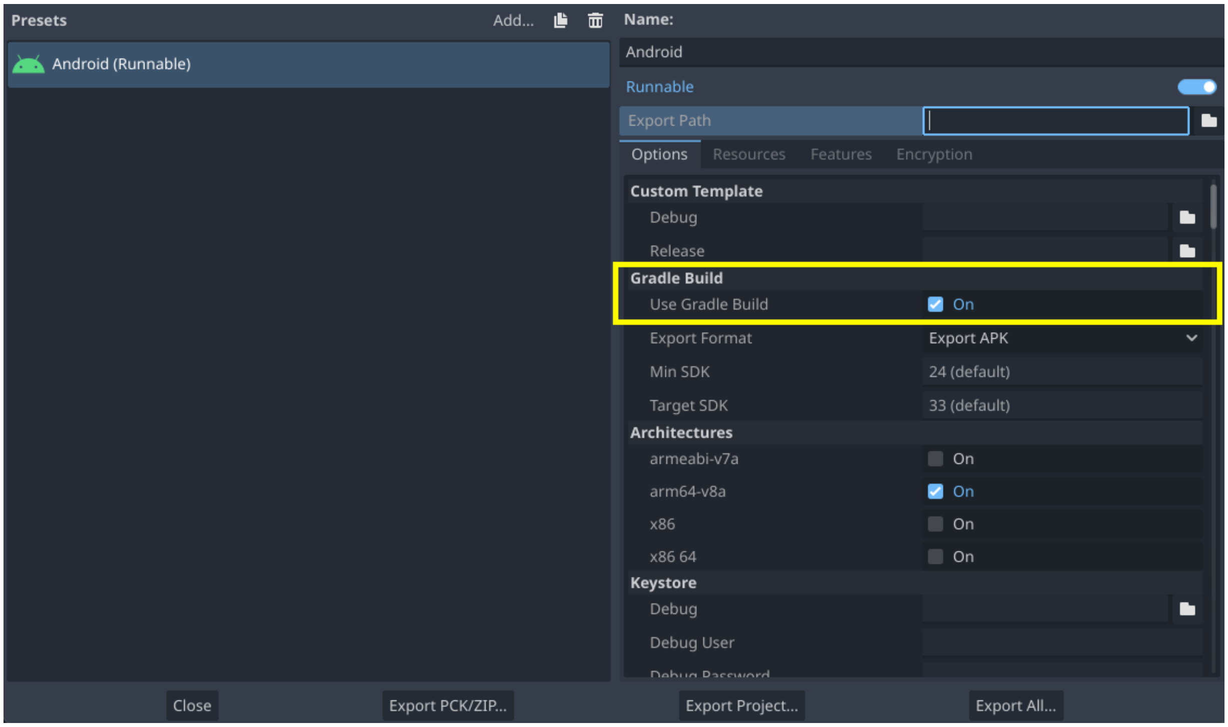

Project
Debug
Editor
Help
Project Settings...
Version Control
Export...
Install Android Build Template...
Open Project Data Folder
Tools
Quit to Project List
Control+Shift+Q

Make sure export templates are downloaded. If not, this menu will help you download them.

A Gradle-based Android project will be created under `res://android/build`. Editing these files is not needed unless you really need to modify the project.

Enabling the gradle build and exporting

When setting up the Android project in the **Project > Export** dialog, **Gradle Build** needs to be enabled:


Presets
Add...
Android (Runnable)
Options
Resources
Features
Encryption
Close
Export PCK/ZIP...
Export Project...
Export All...

Name: Android
Runnable
Export Path

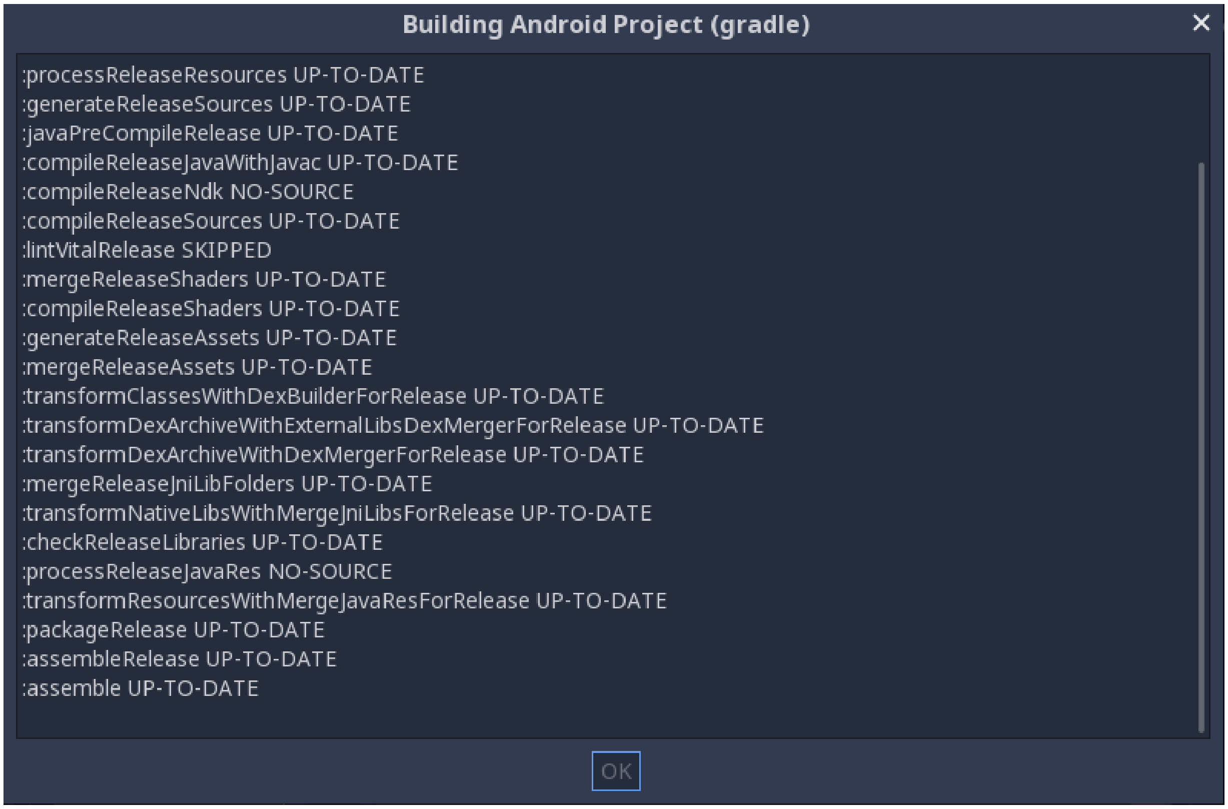
Custom Template
Debug
Release

Gradle Build
Use Gradle Build ☑ On
Export Format: Export APK
Min SDK: 24 (default)
Target SDK: 33 (default)

Architectures
armeabi-v7a ☐ On
arm64-v8a ☑ On
x86 ☐ On
x86 64 ☐ On

Keystore
Debug
Debug User
Debug Password

From now on, attempting to export the project or one-click deploy will call the Gradle [https://gradle.org/] build system to generate fresh templates (this window will appear every time):



```
:processReleaseResources UP-TO-DATE
:generateReleaseSources UP-TO-DATE
:javaPreCompileRelease UP-TO-DATE
:compileReleaseJavaWithJavac UP-TO-DATE
:compileReleaseNdk NO-SOURCE
:compileReleaseSources UP-TO-DATE
:lintVitalRelease SKIPPED
:mergeReleaseShaders UP-TO-DATE
:compileReleaseShaders UP-TO-DATE
:generateReleaseAssets UP-TO-DATE
:mergeReleaseAssets UP-TO-DATE
:transformClassesWithDexBuilderForRelease UP-TO-DATE
:transformDexArchiveWithExternalLibsDexMergerForRelease UP-TO-DATE
:transformDexArchiveWithDexMergerForRelease UP-TO-DATE
:mergeReleaseJniLibFolders UP-TO-DATE
:transformNativeLibsWithMergeJniLibsForRelease UP-TO-DATE
:checkReleaseLibraries UP-TO-DATE
:processReleaseJavaRes NO-SOURCE
:transformResourcesWithMergeJavaResForRelease UP-TO-DATE
:packageRelease UP-TO-DATE
:assembleRelease UP-TO-DATE
:assemble UP-TO-DATE
```

The templates built will be used automatically afterwards, so no further configuration is needed.

Note

When using the gradle Android build system, assets that are placed within a folder whose name begins with an underscore will not be included in the generated APK. This does not apply to assets whose *file* name begins with an underscore.

For example, `_example/image.png` will **not** be included as an asset, but `_image.png` will.

Exporting for the Web

See also

This page describes how to export a Godot project to HTML5. If you're looking to compile editor or export template binaries from source instead, read Compiling for the Web.

HTML5 export allows publishing games made in Godot Engine to the browser. This requires support for WebAssembly [https://webassembly.org/], WebGL [https://www.khronos.org/webgl/] and SharedArrayBuffer [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer] in the user's browser.

Attention

Projects written in C# using Godot 4 currently cannot be exported to the web. To use C# on web platforms, use Godot 3 instead.

Tip

Use the browser-integrated developer console, usually opened with `F12` (`Cmd + Option + I` on macOS), to view **debug information** like JavaScript, engine, and WebGL errors.

Attention

Godot 4's HTML5 exports currently cannot run on macOS and iOS due to upstream bugs with SharedArrayBuffer and WebGL 2.0. We recommend using macOS and iOS native export functionality instead, as it will also result in better performance.

Godot 3's HTML5 exports are more compatible with various browsers in general, especially when using the GLES2 rendering backend (which only requires WebGL 1.0).

WebGL version

Godot 4.0 and later can only target WebGL 2.0 (using the Compatibility rendering method). There is no stable way to run Vulkan applications on the web yet.

See Can I use WebGL 2.0 [https://caniuse.com/webgl2] for a list of browser versions supporting WebGL 2.0. Note that Safari has several issues with WebGL 2.0 support that other browsers don't have, so we recommend using a Chromium-based browser or Firefox if possible.

Export options

If a runnable web export template is available, a button appears between the *Stop scene* and *Play edited Scene* buttons in the editor to quickly open the game in the default browser for testing.

If your project uses GDExtension **Extension Support** needs to be enabled.

If you plan to use VRAM compression make sure that **Vram Texture Compression** is enabled for the targeted platforms (enabling both **For Desktop** and **For Mobile** will result in a bigger, but more compatible export).

If a path to a **Custom HTML shell** file is given, it will be used instead of the default HTML page. See Custom HTML page for Web export.

Head Include is appended into the `<head>` element of the generated HTML page. This allows to, for example, load webfonts and third-party JavaScript APIs, include CSS, or run JavaScript code.

Important

Each project must generate their own HTML file. On export, several text placeholders are replaced in the generated HTML file specifically for the given export options. Any direct modifications to that HTML file will be lost in future exports. To customize the generated file, use the **Custom HTML shell** option.

Limitations

For security and privacy reasons, many features that work effortlessly on native platforms are more complicated on the web platform. Following is a list of limitations you should be aware of when porting a Godot game to the web.

Important

Browser vendors are making more and more functionalities only available in secure contexts [https://developer.mozilla.org/en-US/docs/Web/Security/Secure_Contexts], this means that such features are only be available if the web page is served via a secure HTTPS connection (localhost is usually exempt from such requirement).

Using cookies for data persistence

Users must **allow cookies** (specifically IndexedDB) if persistence of the `user://` file system is desired. When playing a game presented in an `iframe`, **third-party** cookies must also be enabled. Incognito/private browsing mode also prevents persistence.

The method `OS.is_userfs_persistent()` can be used to check if the `user://` file system is persistent, but can give false positives in some cases.

Background processing

The project will be paused by the browser when the tab is no longer the active tab in the user's browser. This means functions such as `_process()` and `_physics_process()` will no longer run until the tab is made active again by the user (by switching back to the tab). This can cause networked games to disconnect if the user switches tabs for a long duration.

This limitation does not apply to unfocused browser *windows*. Therefore, on the user's side, this can be worked around by running the project in a separate *window* instead of a separate tab.

Full screen and mouse capture

Browsers do not allow arbitrarily **entering full screen**. The same goes for **capturing the cursor**. Instead, these actions have to occur as a response to a JavaScript input event. In Godot, this means entering full screen from within a pressed input event callback such as `_input` or `_unhandled_input`. Querying the Input singleton is not sufficient, the relevant input event must currently be active.

For the same reason, the full screen project setting doesn't work unless the engine is started from within a valid input event handler. This requires customization of the HTML page.

Audio

Some browsers restrict autoplay for audio on websites. The easiest way around this limitation is to request the player to click, tap or press a key/button to enable audio, for instance when displaying a splash screen at the start of your game.

See also

Google offers additional information about their Web Audio autoplay policies [https://sites.google.com/a/chromium.org/dev/audio-video/autoplay].

Apple's Safari team also posted additional information about their Auto-Play Policy Changes for macOS [https://webkit.org/blog/7734/auto-play-policy-changes-for-macos/].

Warning

Access to microphone requires a secure context.

Networking

Low level networking is not implemented due to lacking support in browsers.

Currently, only HTTP client, HTTP requests, WebSocket (client) and WebRTC are supported.

The HTTP classes also have several restrictions on the HTML5 platform:

- Accessing or changing the `StreamPeer` is not possible
- Threaded/Blocking mode is not available

- Cannot progress more than once per frame, so polling in a loop will freeze
- No chunked responses
- Host verification cannot be disabled
- Subject to same-origin policy [https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy]

Clipboard

Clipboard synchronization between engine and the operating system requires a browser supporting the Clipboard API [https://developer.mozilla.org/en-US/docs/Web/API/Clipboard_API], additionally, due to the API asynchronous nature might not be reliable when accessed from GDScript.

Warning

Requires a secure context.

Gamepads

Gamepads will not be detected until one of their button is pressed. Gamepads might have the wrong mapping depending on the browser/OS/gamepad combination, sadly the Gamepad API [https://developer.mozilla.org/en-US/docs/Web/API/Gamepad_API/Using_the_Gamepad_API] does not provide a reliable way to detect the gamepad information necessary to remap them based on model/vendor/OS due to privacy considerations.

Warning

Requires a secure context.

Boot splash is not displayed

The default HTML page does not display the boot splash while loading. However, the image is exported as a PNG file, so custom HTML pages can display it.

Serving the files

Exporting for the web generates several files to be served from a web server, including a default HTML page for presentation. A custom HTML file can be used, see Custom HTML page for Web export.

Warning

To ensure low audio latency and the ability to use Thread in web exports, Godot 4 web exports always use SharedArrayBuffer [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer]. This requires a secure context, while also requiring the following CORS headers to be set when serving the files:

```
Cross-Origin-Opener-Policy: same-origin
Cross-Origin-Embedder-Policy: require-corp
```

If you don't control the web server or are unable to add response headers, use coi-serviceworker [https://github.com/gzuidhof/coi-serviceworker] as a workaround.

If the client doesn't receive the required response headers, **the project will not run**.

The generated `.html` file can be used as `DirectoryIndex` in Apache servers and can be renamed to e.g. `index.html` at any time. Its name is never depended on by default.

The HTML page draws the game at maximum size within the browser window. This way, it can be inserted into an `<iframe>` with the game's size, as is common on most web game hosting sites.

The other exported files are served as they are, next to the `.html` file, names unchanged. The `.wasm` file is a binary WebAssembly module implementing the engine. The `.pck` file is the Godot main pack containing your game. The `.js` file contains start-up code and is used by the `.html` file to access the engine. The `.png` file contains the boot splash image. It is not used in the default HTML page, but is included for custom HTML pages.

The `.pck` file is binary, usually delivered with the MIME-type *application/octet-stream*. The `.wasm` file is delivered as *application/wasm*.

Warning

Delivering the WebAssembly module (`.wasm`) with a MIME-type other than *application/wasm* can prevent some start-up optimizations.

Delivering the files with server-side compression is recommended especially for the `.pck` and `.wasm` files, which are usually large in size. The WebAssembly module compresses particularly well, down to around a quarter of its original size with gzip compression. Consider using Brotli precompression if supported on your web server for further file size savings.

Hosts that provide on-the-fly compression: GitHub Pages (gzip)

Hosts that don't provide on-the-fly compression: itch.io, GitLab Pages (supports manual gzip precompression [https://webd97.de/post/gitlab-pages-compression/])

Tip

The Godot repository includes a Python script to host a local web server [https://raw.githubusercontent.com/godotengine/godot/master/platform/web/serve.py]. This script is intended for testing the web editor, but it can also be used to test exported projects.

Save the linked script to a file called `serve.py`, move this file to the folder containing the exported project's `index.html`, then run the following command in a command prompt within the same folder:

```
# You may need to replace `python` with `python3` on some platforms.
python serve.py --root .
```

On Windows, you can open a command prompt in the current folder by holding `Shift` and right-clicking on empty space in Windows Explorer, then choosing **Open PowerShell window here**.

This will serve the contents of the current folder and open the default web browser automatically.

Note that for production use cases, this Python-based web server should not be used. Instead, you should use an established web server such as Apache or nginx.

Interacting with the browser and JavaScript

See the dedicated page on how to interact with JavaScript and access some unique Web browser features.

Environment variables

You can use the following environment variables to set export options outside of the editor. During the export process, these override the values that you set in the export menu.

HTML5 export environment variables

Export option	Environment variable
Encryption / Encryption Key	`GODOT_SCRIPT_ENCRYPTION_KEY`

Exporting for dedicated servers

If you want to run a dedicated server for your project on a machine that doesn't have a GPU or display server available, you'll need to run Godot with the `headless` display server and `Dummy` audio driver.

Since Godot 4.0, this can be done by running a Godot binary on any platform with the `--headless` command line argument, or running a project exported as dedicated server. You do not need to use a specialized server binary anymore, unlike Godot 3.x.

Editor versus export template

It is possible to use either an editor or export template (debug or release) binary in headless mode. Which one you should use depends on your use case:

- **Export template:** Use this one for running dedicated servers. It does not contain editor functionality, and is therefore smaller and more optimized.
- **Editor:** This binary contains editor functionality and is intended to be used for exporting projects. This binary *can* be used to run dedicated servers, but it's not recommended as it's larger and less optimized.

Export approaches

There are two ways to export a project for a server:

- Create a separate export preset for the platform that will host the server, then export your project as usual.
- Export a PCK file only, preferably for the platform that matches the platform that will host the server. Place this PCK file in the same folder as an export template binary, rename the binary to have the same name as the PCK (minus the file extension), then run the binary.

Both methods should result in identical output. The rest of the page will focus on the first approach.

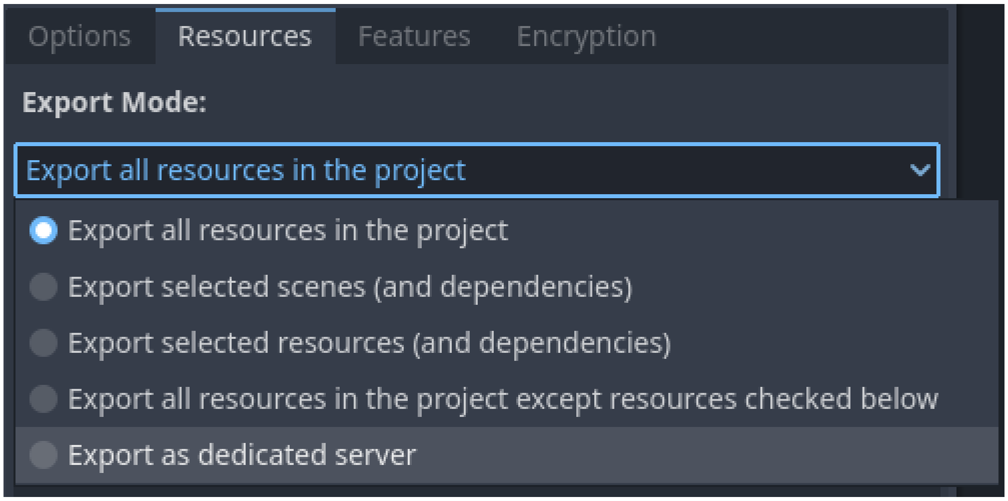
See Exporting projects for more information.

Exporting a project for a dedicated server

If you export a project as usual when targeting a server, you will notice that the PCK file is just as large as for the client. This is because it includes all resources, including those the server doesn't need (such as texture data). Additionally, headless mode won't be automatically used; the user will have to specify `--headless` to make sure no window spawns.

Many resources such as textures can be stripped from the PCK file to greatly reduce its size. Godot offers a way to do this for textures and materials in a way that preserves references in scene or resource files (built-in or external).

To begin doing so, make sure you have a dedicated export preset for your server, then select it, go to its **Resources** tab and change its export mode:



Export Mode:

Export all resources in the project

- ● Export all resources in the project
- ○ Export selected scenes (and dependencies)
- ○ Export selected resources (and dependencies)
- ○ Export all resources in the project except resources checked below
- ○ Export as dedicated server

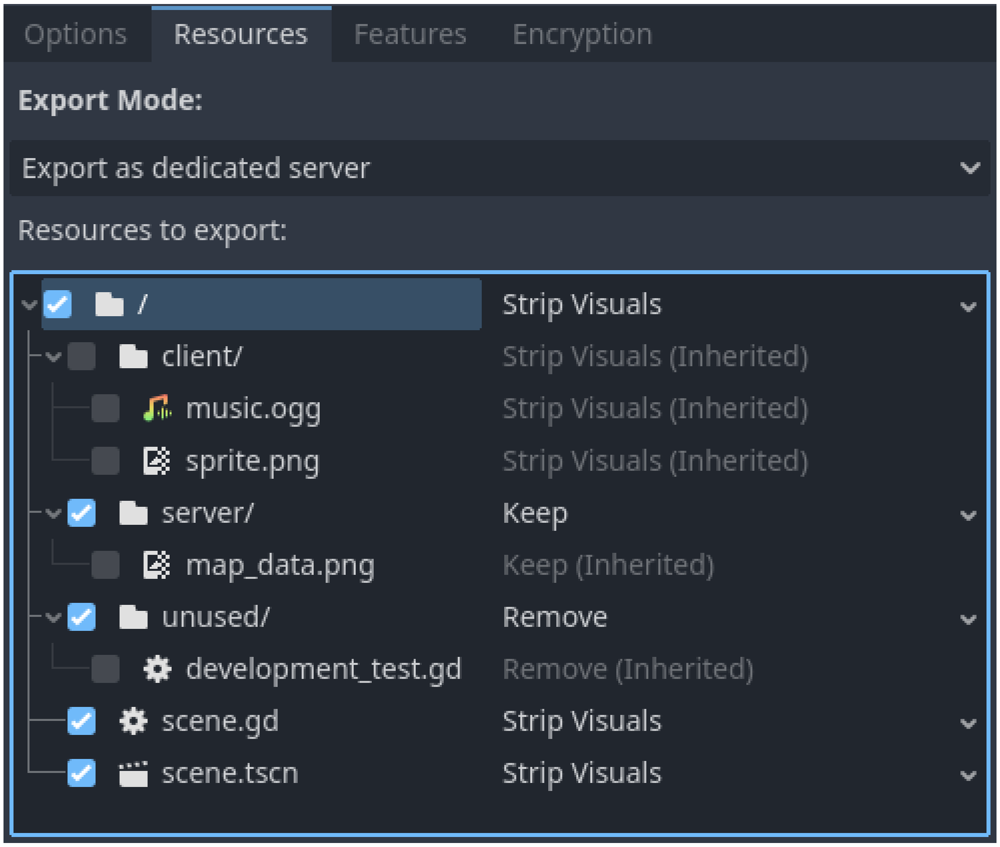
Choosing the **Export as dedicated server** export mode in the export preset

When this export mode is chosen, the `dedicated_server` feature tag is automatically added to the exported project.

Note

If you do not wish to use this export mode but still want the feature tag, you can write the name `dedicated_server` in the **Features** tab of the export preset. This will also force `--headless` when running the exported project.

After selecting this export mode, you will be presented with a list of resources in the project:



Export Mode:

Export as dedicated server

Resources to export:

- ☑ / — Strip Visuals
  - ☐ client/ — Strip Visuals (Inherited)
    - ☐ music.ogg — Strip Visuals (Inherited)
    - ☐ sprite.png — Strip Visuals (Inherited)
  - ☑ server/ — Keep
    - ☐ map_data.png — Keep (Inherited)
  - ☑ unused/ — Remove
    - ☐ development_test.gd — Remove (Inherited)
  - ☑ scene.gd — Strip Visuals
  - ☑ scene.tscn — Strip Visuals

Choosing resources to keep, keep with stripped visuals or remove

Ticking a box allows you to override options for the specified file or folder. Checking boxes does **not** affect which files are exported; this is done by the options selected for each checkbox instead.

Files within a checked folder will automatically use the parent's option by default, which is indicated by the **(Inherited)** suffix for the option name (and the option name being grayed out). To change the option for a file whose option is currently inherited, you must tick the box next to it first.

- **Strip Visuals:** Export this resource, with visual files (textures and materials) replaced by placeholder classes. Placeholder classes store the image size (as it's sometimes used to position elements in a 2D scene), but nothing else.
- **Keep:** Export this resource as usual, with visual files intact.

- **Remove:** The file is not included in the PCK. This is useful to ignore scenes and resources that only the client needs. If you do so, make sure the server doesn't reference these client-only scenes and resources in any way.

The general recommendation is to use **Strip Visuals** whenever possible, unless the server needs to access image data such as pixels' colors. For example, if your server generates collision data based on an image's contents, you need to use **Keep** for that particular image.

Tip

To check the file structure of your exported PCK, use the **Export PCK/ZIP...** button with a `.zip` file extension, then open the resulting ZIP file in a file manager.

Warning

Be careful when using the **Remove** mode, as scenes/resources that reference a removed file will no longer be able to load successfully.

If you wish to remove specific resources but make the scenes still be able to load without them, you'll have to remove the reference in the scene file and load the files to the nodes' properties using `load()` in a script. This approach can be used to strip resources that Godot doesn't support replacing with placeholders yet, such as audio.

Removing textures is often what makes the greatest impact on the PCK size, so it is recommended to stick with **Strip Visuals** at first.

With the above options used, a PCK for the client (which exports all resources normally) will look as follows:

```
.
├── .godot
│   ├── exported
│   │   └── 133200997
│   │       └── export-78c237d4bfdb4e1d02e0b5f38ddfd8bd-scene.scn
│   ├── global_script_class_cache.cfg
│   ├── imported
│   │   ├── map_data.png-ce840618f399a990343bfc7298195a13.ctex
│   │   ├── music.ogg-fa883da45ae49695a3d022f64e60aee2.oggvorbisstr
│   │   └── sprite.png-7958af25f91bb9dbae43f35388f8e840.ctex
│   └── uid_cache.bin
├── client
│   ├── music.ogg.import
│   └── sprite.png.import
├── server
│   └── map_data.png.import
├── test
│   └── scene.gd
└── unused
│   └── development_test.gd
├── project.binary
├── scene.gd
├── scene.tscn.remap
```

The PCK's file structure for the server will look as follows:

```
.
├── .godot
│   ├── exported
│   │   └── 3400186661
│   │       ├── export-78c237d4bfdb4e1d02e0b5f38ddfd8bd-scene.scn
│   │       ├── export-7958af25f91bb9dbae43f35388f8e840-sprite.res  # Placeholder texture
│   │       └── export-fa883da45ae49695a3d022f64e60aee2-music.res
│   ├── global_script_class_cache.cfg
│   ├── imported
│   │   └── map_data.png-ce840618f399a990343bfc7298195a13.ctex
│   └── uid_cache.bin
├── client
│   ├── music.ogg.import
│   └── sprite.png.import  # Points to placeholder texture
└── server
│   └── map_data.png.import
├── project.binary
├── scene.gd
├── scene.tscn.remap
```

Starting the dedicated server

If both your client and server are part of the same Godot project, you will have to add a way to start the server directly using a command-line argument.

If you exported the project using the **Export as dedicated server** export mode (or have added `dedicated_server` as a custom feature tag), you can use the `dedicated_server` feature tag to detect whether a dedicated server PCK is being used:

GDScript

```
# Note: Feature tags are case-sensitive.
if OS.has_feature("dedicated_server"):
    # Run your server startup code here...
    pass
```

C#

```
// Note: Feature tags are case-sensitive.
if (OS.HasFeature("dedicated_server"))
{
    // Run your server startup code here...
}
```

If you also wish to host a server when using the built-in `--headless` command line argument, this can be done by adding the following code snippet in your main scene (or an autoload)'s `_ready()` method:

GDScript

```
if DisplayServer.get_name() == "headless":
    # Run your server startup code here...
    #
    # Using this check, you can start a dedicated server by
running
    # a Godot binary (editor or export template) with the `--
headless`
    # command-line argument.
    pass
```

C#

```
using System.Linq;

if (DisplayServer.GetName() == "headless")
{
    // Run your server startup code here...
    //
    // Using this check, you can start a dedicated server by
running
    // a Godot binary (editor or export template) with the `--
headless`
    // command-line argument.
}
```

If you wish to use a custom command line argument, this can be done by adding the following code snippet in your main scene (or an autoload)'s `_ready()` method:

GDScript

```
if "--server" in OS.get_cmdline_user_args():
    # Run your server startup code here...
    #
    # Using this check, you can start a dedicated server by
running
    # a Godot binary (editor or export template) with the `--
server`
    # command-line argument.
    pass
```

C#

```
using System.Linq;

if (OS.GetCmdlineUserArgs().Contains("--server"))
{
    // Run your server startup code here...
    //
    // Using this check, you can start a dedicated server by
running
    // a Godot binary (editor or export template) with the `--
server`
    // command-line argument.
}
```

It's a good idea to add at least one of the above command-line arguments to start a server, as it can be used to test server functionality from the command line without having to export the project.

If your client and server are separate Godot projects, your server should most likely be configured in a way where running the main scene starts a server automatically.

Next steps

On Linux, to make your dedicated server restart after a crash or system reboot, you can create a systemd service [https://medium.com/@benmorel/creating-a-linux-service-with-systemd-

611b5c8b91d6]. This also lets you view server logs in a more convenient fashion, with automatic log rotation provided by systemd.

If you have experience with containers, you could also look into wrapping your dedicated server in a Docker [https://www.docker.com/] container. This way, it can be used more easily in an automatic scaling setup (which is outside the scope of this tutorial).

One-click deploy

What is one-click deploy?

One-click deploy is a feature that is available once a platform is properly configured and a supported device is connected to the computer. Since things can go wrong at many levels (platform may not be configured correctly, SDK may be incorrectly installed, device may be improperly configured, etc.), it's good to let the user know that it exists.

After adding an Android export preset marked as Runnable, Godot can detect when a USB device is connected to the computer and offer the user to automatically export, install and run the project (in debug mode) on the device. This feature is called *one-click deploy*.

Note

One-click deploy is only available once you've added an export template marked as **Runnable** in the Export dialog. You can mark several export presets as runnable, but only one preset per platform may be marked as runnable. If you mark a second preset in a given platform as runnable, the other preset will no longer be marked as runnable.

Supported platforms

- **Android:** Exports the project with debugging enabled and runs it on the connected device.

  - Make sure to follow the steps described in Exporting for Android. Otherwise, the one-click deploy button won't appear.
  - If you have more than one device connected, Godot will ask you which device the project should be exported to.

- **iOS:** Exports the project with debugging enabled and runs it on the connected device or simulator.

  - Make sure to follow the steps described in Exporting for iOS. Otherwise, the one-click deploy button won't appear.
  - For each new bundle identifier, export the project, open it in the Xcode, and build at least once to create new provisioning profile or create a provisioning profile in the Apple Developer account dashboard.
  - If you have more than one device connected, Godot will ask you which device the project should be exported to.

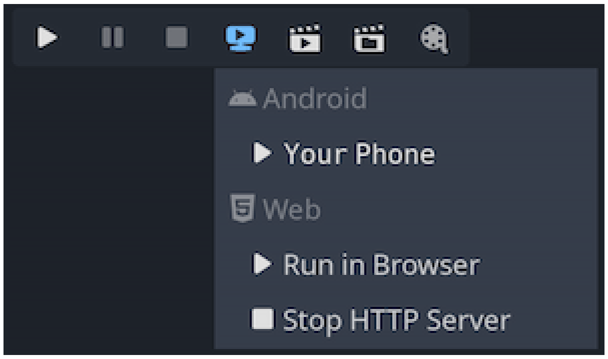
- **Desktop platforms:** Exports the project with debugging enabled and runs it on the remote computer via SSH.

- **Web:** Starts a local web server and runs the exported project by opening the default web browser. This is only accessible on `localhost` by default. See Troubleshooting for making the exported project accessible on remote devices.

Using one-click deploy

- **Android:**
  - Enable developer mode on your mobile device then enable USB debugging in the device's settings.

    - After enabling USB debugging, connect the device to your PC using a USB cable.
    - For advanced users, it should also be possible to use wireless ADB.
- **iOS:**
    - Install Xcode, accept Xcode license and login with your Apple Developer account.
    - If you are using Xcode 14 or earlier, install ios-deploy [https://github.com/ios-control/ios-deploy] and set path to *ios-deploy* in the Editor Settings (see *Export → iOS → iOS Deploy*).
    - For running on device:
        - Pair your mobile device with a Mac.
        - Enable developer mode on your device.
        - Device can be connected via USB or local network.
        - Make sure the device is on the same local network and a correct network interface is selected in the editor settings (see *Network → Debug → Remote Host*). By default, the editor is listening for *localhost* connections only.
    - For running in simulator:
        - Install and start simulator from the Xcode.
    - Device screen should be unlocked.
- **Desktop platforms:**
    - Enable *SSH Remote Deploy* and configure connection settings in the project export setting.
- Make sure there is an export preset marked as **Runnable** for the target platform (Android, iOS or Web).
- If everything is configured correctly and with no errors, platform-specific icons will appear in the top-right corner of the editor.
- Click the button to export to the desired platform in one click.



Troubleshooting

Android

If you can't see the device in the list of devices when running the `adb devices` command in a terminal, it will not be visible by Godot either. To resolve this:

- Check if USB debugging is enabled *and authorized on the device*. Try unlocking your device and accepting the authorization prompt if you see any. If you can't see this prompt, running `adb devices` on your PC should make the authorization prompt appear on the device.
- Try revoking the debugging authorization [https://stackoverflow.com/questions/23081263/adb-android-device-unauthorized] in the device's developer settings, then follow the steps again.
- Try using USB debugging instead of wireless debugging or vice versa. Sometimes, one of those can work better than the other.
- On Linux, you may be missing the required udev rules [https://github.com/M0Rf30/android-udev-rules] for your device to be recognized.

Web

By default, the web server started by the editor is only accessible from `localhost`. This means the web server can't be reached by other devices on the local network or the Internet (if port forwarding is set up on the router). This is done for security reasons, as you may not want other devices to be able to access the exported project while you're testing it. Binding to `localhost` also prevents a firewall popup from appearing when you use one-click deploy for the web platform.

To make the local web server accessible over the local network, you'll need to change the **Export > Web > HTTP Host** editor setting to `0.0.0.0`. You will also need to enable **Export > Web > Use TLS** as SharedArrayBuffer requires the use of a secure connection to work, *unless* connecting to `localhost`. However, since other clients will be connecting to a remote device, the use of TLS is absolutely required here.

To make the local web server accessible over the Internet, you'll also need to forward the **Export > Web > HTTP Port** port specified in the Editor Settings (`8060` by default) in TCP on your router. This is usually done by accessing your router's web interface then adding a NAT rule for the port in question. For IPv6 connections, you should allow the port in the router's IPv6 firewall instead. Like for local network devices, you will also need to enable **Export > Web > Use TLS**.

Note

When **Use TLS** is enabled, you will get a warning from your web browser as Godot will use a temporary self-signed certificate. You can safely ignore it and bypass the warning by clicking **Advanced** and then **Proceed to (address)**.

If you have an SSL/TLS certificate that is trusted by browsers, you can specify the paths to the key and certificate files in the **Export > Web > TLS Key** and **Export > Web > TLS Certificate**. This will only work if the project is accessed through a domain name that is part of the TLS certificate.

File and data I/O

- Background loading
- File paths in Godot projects
- Saving games
- Runtime file loading and saving
- Binary serialization API

Background loading

Commonly, games need to load resources asynchronously. When switching the main scene of your game (e.g. going to a new level), you might want to show a loading screen with some indication that progress is being made, or you may want to load additional resources during gameplay.

The standard load method (ResourceLoader.load or GDScript's simpler load) blocks your thread, making your game appear unresponsive while the resource is being loaded.

One way around this is using `ResourceLoader` to load resources asynchronously in background threads.

Using ResourceLoader

Generally, you queue requests to load resources for a path using ResourceLoader.load_threaded_request, which will then be loaded in threads in the background.

You can check the status with ResourceLoader.load_threaded_get_status. Progress can be obtained by passing an array variable via progress which will return a one element array containing the percentage.

Finally, you retrieve loaded resources by calling ResourceLoader.load_threaded_get.

Once you call `load_threaded_get()`, either the resource finished loading in the background and will be returned instantly or the load will block at this point like `load()` would. If you want to guarantee this does not block, you either need to ensure there is enough time between requesting the load and retrieving the resource or you need to check the status manually.

Example

This example demonstrates how to load a scene in the background. We will have a button spawn an enemy when pressed. The enemy will be `Enemy.tscn` which we will load on `_ready` and instantiate when pressed. The path will be `"Enemy.tscn"` which is located at `res://Enemy.tscn`.

First, we will start a request to load the resource and connect the button:

GDScript

```
const ENEMY_SCENE_PATH : String = "Enemy.tscn"

func _ready():
    ResourceLoader.load_threaded_request(ENEMY_SCENE_PATH)
    self.pressed.connect(_on_button_pressed)
```

C#

```
using Godot;

public partial class MyButton : Button
{
    private const string EnemyScenePath = "Enemy.tscn";

    public override void _Ready()
    {
        ResourceLoader.LoadThreadedRequest(EnemyScenePath);
        Pressed += OnButtonPressed;
    }
}
```

Now `_on_button_pressed` will be called when the button is pressed. This method will be used to spawn an enemy.

GDScript

```
func _on_button_pressed(): # Button was pressed.
    # Obtain the resource now that we need it.
    var enemy_scene =
ResourceLoader.load_threaded_get(ENEMY_SCENE_PATH)
    # Instantiate the enemy scene and add it to the current
scene.
    var enemy = enemy_scene.instantiate()
    add_child(enemy)
```

C#

```
private void OnButtonPressed() // Button was pressed.
{
    // Obtain the resource now that we need it.
    var enemyScene =
(PackedScene)ResourceLoader.LoadThreadedGet(EnemyScenePath);
    // Instantiate the enemy scene and add it to the current
scene.
    var enemy = enemyScene.Instantiate();
    AddChild(enemy);
}
```

File paths in Godot projects

This page explains how file paths work inside Godot projects. You will learn how to access paths in your projects using the `res://` and `user://` notations, and where Godot stores project and editor files on your and your users' systems.

Path separators

To make supporting multiple platforms easier, Godot uses **UNIX-style path separators** (forward slash `/`). These work on all platforms, **including Windows**.

Instead of writing paths like `C:\Projects\Game`, in Godot, you should write `C:/Projects/Game`.

Windows-style path separators (backward slash `\`) are also supported in some path-related methods, but they need to be doubled (`\\`), as `\` is normally used as an escape for characters with a special meaning.

This makes it possible to work with paths returned by other Windows applications. We still recommend using only forward slashes in your own code to guarantee that everything will work as intended.

Tip

The String class offers over a dozen methods to work with strings that represent file paths:

- String.filecasecmp_to()
- String.filenocasecmp_to()

- String.get_base_dir()
- String.get_basename()
- String.get_extension()
- String.get_file()
- String.is_absolute_path()
- String.is_relative_path()
- String.is_valid_filename()
- String.path_join()
- String.simplify_path()
- String.validate_filename()

Accessing files in the project folder (`res://`)

Godot considers that a project exists in any folder that contains a `project.godot` text file, even if the file is empty. The folder that contains this file is your project's root folder.

You can access any file relative to it by writing paths starting with `res://`, which stands for resources. For example, you can access an image file `character.png` located in the project's root folder in code with the following path: `res://character.png`.

Accessing persistent user data (`user://`)

To store persistent data files, like the player's save or settings, you want to use `user://` instead of `res://` as your path's prefix. This is because when the game is running, the project's file system will likely be read-only.

The `user://` prefix points to a different directory on the user's device. Unlike `res://`, the directory pointed at by `user://` is created automatically and *guaranteed* to be writable to, even in an exported project.

The location of the `user://` folder depends on what is configured in the Project Settings:

- By default, the `user://` folder is created within Godot's editor data path in the `app_userdata/[project_name]` folder. This is the default so that prototypes and test projects stay self-contained within Godot's data folder.
- If application/config/use_custom_user_dir is enabled in the Project Settings, the `user://` folder is created **next to** Godot's editor data path, i.e. in the standard location for applications data.
    - By default, the folder name will be inferred from the project name, but it can be further customized with application/config/custom_user_dir_name. This path can contain path separators, so you can use it e.g. to group projects of a given studio with a `Studio Name/Game Name` structure.

On desktop platforms, the actual directory paths for `user://` are:

Type	Location
Default	Windows: `%APPDATA%\Godot\app_userdata\[project_name]` macOS: `~/Library/Application Support/Godot/app_userdata/[project_name]` Linux: `~/.local/share/godot/app_userdata/[project_name]`

Type	Location
Custom dir	Windows: `%APPDATA%\[project_name]` macOS: `~/Library/Application Support/[project_name]` Linux: `~/.local/share/[project_name]`
Custom dir and name	Windows: `%APPDATA%\[custom_user_dir_name]` macOS: `~/Library/Application Support/[custom_user_dir_name]` Linux: `~/.local/share/[custom_user_dir_name]`

`[project_name]` is based on the application name defined in the Project Settings, but you can override it on a per-platform basis using feature tags.

On mobile platforms, this path is unique to the project and is not accessible by other applications for security reasons.

On HTML5 exports, `user://` will refer to a virtual filesystem stored on the device via IndexedDB. (Interaction with the main filesystem can still be performed through the JavaScriptBridge singleton.)

Converting paths to absolute paths or "local" paths

You can use ProjectSettings.globalize_path() to convert a "local" path like `res://path/to/file.txt` to an absolute OS path. For example, ProjectSettings.globalize_path() can be used to open "local" paths in the OS file manager using OS.shell_open() since it only accepts native OS paths.

To convert an absolute OS path to a "local" path starting with `res://` or `user://`, use [ProjectSettings.localize_path()](). This only works for absolute paths that point to files or folders in your project's root or `user://` folders.

Editor data paths

The editor uses different paths for editor data, editor settings, and cache, depending on the platform. By default, these paths are:

Type	Location
Editor data	Windows: `%APPDATA%\Godot\` macOS: `~/Library/Application Support/Godot/` Linux: `~/.local/share/godot/`
Editor settings	Windows: `%APPDATA%\Godot\` macOS: `~/Library/Application Support/Godot/` Linux: `~/.config/godot/`
Cache	Windows: `%TEMP%\Godot\` macOS: `~/Library/Caches/Godot/` Linux: `~/.cache/godot/`

- **Editor data** contains export templates and project-specific data.
- **Editor settings** contains the main editor settings configuration file as well as various other user-specific customizations (editor layouts, feature profiles, script templates, etc.).

- **Cache** contains data generated by the editor, or stored temporarily. It can safely be removed when Godot is closed.

Godot complies with the XDG Base Directory Specification [https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html] on Linux/*BSD. You can override the `XDG_DATA_HOME`, `XDG_CONFIG_HOME` and `XDG_CACHE_HOME` environment variables to change the editor and project data paths.

Note

If you use Godot packaged as a Flatpak [https://flathub.org/apps/details/org.godotengine.Godot], the editor data paths will be located in subfolders in `~/.var/app/org.godotengine.Godot/`.

Self-contained mode

If you create a file called `._sc_` or `_sc_` in the same directory as the editor binary (or in *MacOS/Contents/* for a macOS editor .app bundle), Godot will enable *self-contained mode*. This mode makes Godot write all editor data, settings, and cache to a directory named `editor_data/` in the same directory as the editor binary. You can use it to create a portable installation of the editor.

The Steam release of Godot [https://store.steampowered.com/app/404790/] uses self-contained mode by default.

Note

Self-contained mode is not supported in exported projects yet. To read and write files relative to the executable path, use OS.get_executable_path(). Note that writing files in the executable path only works if the executable is placed in a writable location (i.e. **not** Program Files or another directory that is read-only for regular users).

Saving games

Introduction

Save games can be complicated. For example, it may be desirable to store information from multiple objects across multiple levels. Advanced save game systems should allow for additional information about an arbitrary number of objects. This will allow the save function to scale as the game grows more complex.

Note

If you're looking to save user configuration, you can use the ConfigFile class for this purpose.

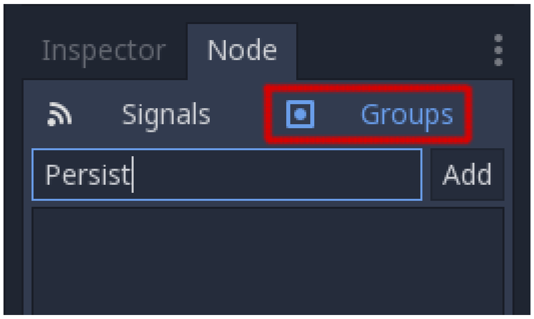
See also

You can see how saving and loading works in action using the Saving and Loading (Serialization) demo project [https://github.com/godotengine/godot-demo-projects/blob/master/loading/serialization].

Identify persistent objects

Firstly, we should identify what objects we want to keep between game sessions and what information we want to keep from those objects. For this tutorial, we will use groups to mark and handle objects to be saved, but other methods are certainly possible.

We will start by adding objects we wish to save to the "Persist" group. We can do this through either the GUI or script. Let's add the relevant nodes using the GUI:



Once this is done, when we need to save the game, we can get all objects to save them and then tell them all to save with this script:

GDScript

```
var save_nodes = get_tree().get_nodes_in_group("Persist")
for node in save_nodes:
    # Now, we can call our save function on each node.
```

C#

```
var saveNodes = GetTree().GetNodesInGroup("Persist");
foreach (Node saveNode in saveNodes)
{
    // Now, we can call our save function on each node.
}
```

Serializing

The next step is to serialize the data. This makes it much easier to read from and store to disk. In this case, we're assuming each member of group Persist is an instanced node and thus has a path. GDScript has helper class JSON to convert between dictionary and string, Our node needs to contain a save function that returns this data. The save function will look like this:

GDScript

```
func save():
    var save_dict = {
        "filename" : get_scene_file_path(),
        "parent" : get_parent().get_path(),
        "pos_x" : position.x, # Vector2 is not supported by JSON
        "pos_y" : position.y,
        "attack" : attack,
        "defense" : defense,
        "current_health" : current_health,
        "max_health" : max_health,
        "damage" : damage,
        "regen" : regen,
        "experience" : experience,
        "tnl" : tnl,
        "level" : level,
        "attack_growth" : attack_growth,
        "defense_growth" : defense_growth,
        "health_growth" : health_growth,
        "is_alive" : is_alive,
        "last_attack" : last_attack
    }
    return save_dict
```

C#

```
public Godot.Collections.Dictionary<string, Variant> Save()
{
    return new Godot.Collections.Dictionary<string, Variant>()
    {
        { "Filename", SceneFilePath },
        { "Parent", GetParent().GetPath() },
        { "PosX", Position.X }, // Vector2 is not supported by JSON
        { "PosY", Position.Y },
        { "Attack", Attack },
        { "Defense", Defense },
        { "CurrentHealth", CurrentHealth },
        { "MaxHealth", MaxHealth },
        { "Damage", Damage },
        { "Regen", Regen },
        { "Experience", Experience },
        { "Tnl", Tnl },
        { "Level", Level },
        { "AttackGrowth", AttackGrowth },
        { "DefenseGrowth", DefenseGrowth },
        { "HealthGrowth", HealthGrowth },
        { "IsAlive", IsAlive },
        { "LastAttack", LastAttack }
    };
}
```

This gives us a dictionary with the style `{ "variable_name":value_of_variable }`, which will be useful when loading.

Saving and reading data

As covered in the File system tutorial, we'll need to open a file so we can write to it or read from it. Now that we have a way to call our groups and get their relevant data, let's use the class JSON to convert it into an easily stored string and store them in a file. Doing it this way ensures that each line is its own object, so we have an easy way to pull the data out of the file as well.

GDScript

```
# Note: This can be called from anywhere inside the tree. This
function is
# path independent.
# Go through everything in the persist category and ask them to
return a
# dict of relevant variables.
func save_game():
    var save_file = FileAccess.open("user://savegame.save",
FileAccess.WRITE)
    var save_nodes = get_tree().get_nodes_in_group("Persist")
    for node in save_nodes:
        # Check the node is an instanced scene so it can be
instanced again during load.
        if node.scene_file_path.is_empty():
            print("persistent node '%s' is not an instanced
scene, skipped" % node.name)
            continue

        # Check the node has a save function.
        if !node.has_method("save"):
            print("persistent node '%s' is missing a save()
function, skipped" % node.name)
            continue

        # Call the node's save function.
        var node_data = node.call("save")

        # JSON provides a static method to serialized JSON
string.
        var json_string = JSON.stringify(node_data)

        # Store the save dictionary as a new line in the save
file.
        save_file.store_line(json_string)
```

C#

```
// Note: This can be called from anywhere inside the tree. This
function is
// path independent.
// Go through everything in the persist category and ask them
to return a
// dict of relevant variables.
public void SaveGame()
{
    using var saveFile =
FileAccess.Open("user://savegame.save",
FileAccess.ModeFlags.Write);

    var saveNodes = GetTree().GetNodesInGroup("Persist");
    foreach (Node saveNode in saveNodes)
    {
        // Check the node is an instanced scene so it can be
instanced again during load.
        if (string.IsNullOrEmpty(saveNode.SceneFilePath))
        {
            GD.Print($"persistent node '{saveNode.Name}' is not
an instanced scene, skipped");
            continue;
        }

        // Check the node has a save function.
        if (!saveNode.HasMethod("Save"))
        {
            GD.Print($"persistent node '{saveNode.Name}' is
missing a Save() function, skipped");
            continue;
        }

        // Call the node's save function.
        var nodeData = saveNode.Call("Save");

        // Json provides a static method to serialized JSON
string.
        var jsonString = Json.Stringify(nodeData);

        // Store the save dictionary as a new line in the save
file.
        saveFile.StoreLine(jsonString);
    }
}
```

Game saved! Now, to load, we'll read each line. Use the parse method to read the JSON string back to a dictionary, and then iterate over the dict to read our values. But we'll need to first create the object and we can use the filename and parent values to achieve that. Here is our load function:

GDScript

```
# Note: This can be called from anywhere inside the tree. This
function
# is path independent.
func load_game():
    if not FileAccess.file_exists("user://savegame.save"):
        return # Error! We don't have a save to load.

    # We need to revert the game state so we're not cloning
objects
    # during loading. This will vary wildly depending on the
needs of a
    # project, so take care with this step.
    # For our example, we will accomplish this by deleting
saveable objects.
    var save_nodes = get_tree().get_nodes_in_group("Persist")
    for i in save_nodes:
        i.queue_free()

    # Load the file line by line and process that dictionary to
restore
    # the object it represents.
    var save_file = FileAccess.open("user://savegame.save",
FileAccess.READ)
    while save_file.get_position() < save_file.get_length():
        var json_string = save_file.get_line()

        # Creates the helper class to interact with JSON.
        var json = JSON.new()

        # Check if there is any error while parsing the JSON
string, skip in case of failure.
        var parse_result = json.parse(json_string)
        if not parse_result == OK:
            print("JSON Parse Error: ",
json.get_error_message(), " in ", json_string, " at line ",
json.get_error_line())
            continue

        # Get the data from the JSON object.
        var node_data = json.data

        # Firstly, we need to create the object and add it to
the tree and set its position.
        var new_object =
load(node_data["filename"]).instantiate()
        get_node(node_data["parent"]).add_child(new_object)
        new_object.position = Vector2(node_data["pos_x"],
node_data["pos_y"])

        # Now we set the remaining variables.
        for i in node_data.keys():
            if i == "filename" or i == "parent" or i == "pos_x"
or i == "pos_y":
                continue
            new_object.set(i, node_data[i])
```

C#

```
// Note: This can be called from anywhere inside the tree. This
function is
// path independent.
public void LoadGame()
{
    if (!FileAccess.FileExists("user://savegame.save"))
    {
        return; // Error! We don't have a save to load.
    }

    // We need to revert the game state so we're not cloning
objects during loading.
    // This will vary wildly depending on the needs of a
project, so take care with
    // this step.
    // For our example, we will accomplish this by deleting
saveable objects.
    var saveNodes = GetTree().GetNodesInGroup("Persist");
    foreach (Node saveNode in saveNodes)
    {
        saveNode.QueueFree();
    }

    // Load the file line by line and process that dictionary
to restore the object
    // it represents.
    using var saveFile =
FileAccess.Open("user://savegame.save",
FileAccess.ModeFlags.Read);

    while (saveFile.GetPosition() < saveFile.GetLength())
    {
        var jsonString = saveFile.GetLine();

        // Creates the helper class to interact with JSON.
        var json = new Json();
        var parseResult = json.Parse(jsonString);
        if (parseResult != Error.Ok)
        {
            GD.Print($"JSON Parse Error:
{json.GetErrorMessage()} in {jsonString} at line
{json.GetErrorLine()}");
            continue;
        }

        // Get the data from the JSON object.
        var nodeData = new Godot.Collections.Dictionary<string,
Variant>((Godot.Collections.Dictionary)json.Data);

        // Firstly, we need to create the object and add it to
the tree and set its position.
        var newObjectScene = GD.Load<PackedScene>
(nodeData["Filename"].ToString());
        var newObject = newObjectScene.Instantiate<Node>();

GetNode(nodeData["Parent"].ToString()).AddChild(newObject);
        newObject.Set(Node2D.PropertyName.Position, new
Vector2((float)nodeData["PosX"], (float)nodeData["PosY"]));

        // Now we set the remaining variables.
        foreach (var (key, value) in nodeData)
        {
            if (key == "Filename" || key == "Parent" || key ==
"PosX" || key == "PosY")
            {
                continue;
            }
            newObject.Set(key, value);
        }
    }
}
```

Now we can save and load an arbitrary number of objects laid out almost anywhere across the scene tree! Each object can store different data depending on what it needs to save.

Some notes

We have glossed over setting up the game state for loading. It's ultimately up to the project creator where much of this logic goes. This is often complicated and will need to be heavily customized based on the needs of the individual project.

Additionally, our implementation assumes no Persist objects are children of other Persist objects. Otherwise, invalid paths would be created. To accommodate nested Persist objects, consider saving objects in stages. Load parent objects first so they are available for the add_child() call when child objects are loaded. You will also need a way to link children to parents as the NodePath will likely be invalid.

JSON vs binary serialization

For simple game state, JSON may work and it generates human-readable files that are easy to debug.

But JSON has many limitations. If you need to store more complex game state or a lot of it, binary serialization may be a better approach.

JSON limitations

Here are some important gotchas to know about when using JSON.

- **Filesize:** JSON stores data in text format, which is much larger than binary formats.

- **Data types:** JSON only offers a limited set of data types. If you have data types that JSON doesn't have, you will need to translate your data to and from types that JSON can handle. For example, some important types that JSON can't parse are: `Vector2`, `Vector3`, `Color`, `Rect2`, and `Quaternion`.
- **Custom logic needed for encoding/decoding:** If you have any custom classes that you want to store with JSON, you will need to write your own logic for encoding and decoding those classes.

Binary serialization

Binary serialization is an alternative approach for storing game state, and you can use it with the functions `get_var` and `store_var` of FileAccess.

- Binary serialization should produce smaller files than JSON.
- Binary serialization can handle most common data types.
- Binary serialization requires less custom logic for encoding and decoding custom classes.

Note that not all properties are included. Only properties that are configured with the PROPERTY_USAGE_STORAGE flag set will be serialized. You can add a new usage flag to a property by overriding the _get_property_list method in your class. You can also check how property usage is configured by calling `Object._get_property_list`. See PropertyUsageFlags for the possible usage flags.

Runtime file loading and saving

See also

See Saving games for information on saving and loading game progression.

Sometimes, exporting packs, patches, and mods is not ideal when you want players to be able to load user-generated content in your project. It requires users to generate a PCK or ZIP file through the Godot editor, which contains resources imported by Godot.

Example use cases for runtime file loading and saving include:

- Loading texture packs designed for the game.
- Loading user-provided audio tracks and playing them back in an in-game radio station.
- Loading custom levels or 3D models that can be designed with any 3D DCC that can export to glTF (including glTF scenes saved by Godot at runtime).
- Using user-provided fonts for menus and HUD.
- Saving/loading a file format that can contain multiple files but can still easily be read by other applications (ZIP).
- Loading files created by another game or program, or even game data files from another game not made with Godot.

Runtime file loading can be combined with HTTP requests to load resources from the Internet directly.

Warning

Do **not** use this runtime loading approach to load resources that are part of the project, as it's less efficient and doesn't allow benefiting from Godot's resource handling functionality (such as translation remaps). See Import process for details.

See also

You can see how saving and loading works in action using the Runtime File Saving and Loading (Serialization) demo project [https://github.com/godotengine/godot-demo-projects/blob/master/loading/runtime_save_load].

Plain text and binary files

Godot's FileAccess class provides methods to access files on the filesystem for reading and writing:

```
 func save_file(content):
    var file = FileAccess.open("/path/to/file.txt",
FileAccess.WRITE)
    file.store_string(content)

func load_file():
    var file = FileAccess.open("/path/to/file.txt",
FileAccess.READ)
    var content = file.get_as_text()
    return content
```

To handle custom binary formats (such as loading file formats not supported by Godot), FileAccess provides several methods to read/write integers, floats, strings and more. These FileAccess methods have names that start with `get_` and `store_`.

If you need more control over reading binary files or need to read binary streams that are not part of a file, PackedByteArray provides several helper methods to decode/encode series of bytes to integers, floats, strings and more. These PackedByteArray methods have names that start with `decode_` and `encode_`. See also Binary serialization API.

Images

Image's Image.load_from_file static method handles everything, from format detection based on file extension to reading the file from disk.

If you need error handling or more control (such as changing the scale an SVG is loaded at), use one of the following methods depending on the file format:

- Image.load_jpg_from_buffer
- Image.load_ktx_from_buffer
- Image.load_png_from_buffer
- Image.load_svg_from_buffer or Image.load_svg_from_string
- Image.load_tga_from_buffer
- Image.load_webp_from_buffer

Several image formats can also be saved by Godot at runtime using the following methods:

- Image.save_png or Image.save_png_to_buffer
- Image.save_webp or Image.save_webp_to_buffer
- Image.save_jpg or Image.save_jpg_to_buffer
- Image.save_exr or Image.save_exr_to_buffer *(only available in editor builds, cannot be used in exported projects)*

The methods with the `to_buffer` suffix save the image to a PackedByteArray instead of the filesystem. This is useful to send the image over the network or into a ZIP archive without having to write it on the filesystem. This can increase performance by reducing I/O utilization.

Note

If displaying the loaded image on a 3D surface, make sure to call Image.generate_mipmaps so that the texture doesn't look grainy when viewed at a distance. This is also useful in 2D when following instructions on reducing aliasing when downsampling.

Example of loading an image and displaying it in a TextureRect node (which requires conversion to ImageTexture):

```
# Load an image of any format supported by Godot from the
filesystem.
var image = Image.load_from_file(path)
# Optionally, generate mipmaps if displaying the texture on a
3D surface
# so that the texture doesn't look grainy when viewed at a
distance.
#image.generate_mipmaps()
$TextureRect.texture = ImageTexture.create_from_image(image)

# Save the loaded Image to a PNG image.
image.save_png("/path/to/file.png")

# Save the converted ImageTexture to a PNG image.
$TextureRect.texture.get_image().save_png("/path/to/file.png")
```

Audio/video files

Godot supports loading Ogg Vorbis audio at runtime. Note that not *all* files with an `.ogg` extension may be Ogg Vorbis files. Some may be Ogg Theora videos, or contain Opus audio within an Ogg container. These files will **not** load correctly as audio files in Godot.

Example of loading an Ogg Vorbis audio file in an AudioStreamPlayer node:

```
$AudioStreamPlayer.stream =
AudioStreamOggVorbis.load_from_file(path)
```

Example of loading an Ogg Theora video file in a VideoStreamPlayer node:

```
var video_stream_theora = VideoStreamTheora.new()
# File extension is ignored, so it is possible to load Ogg
Theora videos
# that have an `.ogg` extension this way.
video_stream_theora.file = "/path/to/file.ogv"
$VideoStreamPlayer.stream = video_stream_theora

# VideoStreamPlayer's Autoplay property won't work if the
stream is empty
# before this property is set, so call `play()` after setting
`stream`.
$VideoStreamPlayer.play()
```

Note

Godot doesn't support runtime loading of MP3 or WAV files yet. Until this is implemented, it's feasible to implement runtime WAV loading using a script since AudioStreamWAV's `data` property is exposed to scripting.

It's still possible to *save* WAV files using AudioStreamWAV.save_to_wav, which is useful for procedurally generated audio or microphone recordings.

3D scenes

Godot has first-class support for glTF 2.0, both in the editor and exported projects. Using GLTFDocument and GLTFState together, Godot can load and save glTF files in exported projects, in both text (`.gltf`) and binary (`.glb`) formats. The binary format should be preferred as it's faster to write and smaller, but the text format is easier to debug.

Example of loading a glTF scene and appending its root node to the scene:

```
# Load an existing glTF scene.
# GLTFState is used by GLTFDocument to store the loaded scene's
state.
# GLTFDocument is the class that handles actually loading glTF
data into a Godot node tree,
# which means it supports glTF features such as lights and
cameras.
var gltf_document_load = GLTFDocument.new()
var gltf_state_load = GLTFState.new()
var error =
gltf_document_load.append_from_file("/path/to/file.gltf",
gltf_state_load)
if error == OK:
    var gltf_scene_root_node =
gltf_document_load.generate_scene(gltf_state_load)
    add_child(gltf_scene_root_node)
else:
    show_error("Couldn't load glTF scene (error code: %s)." %
error_string(error))

# Save a new glTF scene.
var gltf_document_save := GLTFDocument.new()
var gltf_state_save := GLTFState.new()
gltf_document_save.append_from_scene(gltf_scene_root_node,
gltf_state_save)
# The file extension in the output `path` (`.gltf` or `.glb`)
determines
# whether the output uses text or binary format.
# `GLTFDocument.generate_buffer()` is also available for saving
to memory.
gltf_document_save.write_to_filesystem(gltf_state_save, path)
```

Note

When loading a glTF scene, a *base path* must be set so that external resources like textures can be loaded correctly. When loading from a file, the base path is automatically set to the folder containing the file. When loading from a buffer, this base path must be manually set as there is no way for Godot to infer this path.

To set the base path, set GLTFState.base_path on your GLTFState instance *before* calling GLTFDocument.append_from_buffer or GLTFDocument.append_from_file.

Fonts

FontFile.load_dynamic_font supports the following font file formats: TTF, OTF, WOFF, WOFF2, PFB, PFM

On the other hand, FontFile.load_bitmap_font supports the BMFont [https://www.angelcode.com/products/bmfont/] format (`.fnt` or `.font`).

Additionally, it is possible to load any font that is installed on the system using Godot's support for System fonts.

Example of loading a font file automatically according to its file extension, then adding it as a theme override to a Label node:

```
var path = "/path/to/font.ttf"
var path_lower = path.to_lower()
var font_file = FontFile.new()
if (
        path_lower.ends_with(".ttf")
        or path_lower.ends_with(".otf")
        or path_lower.ends_with(".woff")
        or path_lower.ends_with(".woff2")
        or path_lower.ends_with(".pfb")
        or path_lower.ends_with(".pfm")
):
    font_file.load_dynamic_font(path)
elif path_lower.ends_with(".fnt") or
path_lower.ends_with(".font"):
    font_file.load_bitmap_font(path)
else:
    push_error("Invalid font file format.")

if not font_file.data.is_empty():
    # If font was loaded successfully, add it as a theme
override.
    $Label.add_theme_font_override("font", font_file)
```

ZIP archives

Godot supports reading and writing ZIP archives using the ZIPReader and ZIPPacker classes. This supports any ZIP file, including files generated by Godot's "Export PCK/ZIP" functionality (although these will contain imported Godot resources rather than the original project files).

Note

Use ProjectSettings.load_resource_pack to load PCK or ZIP files exported by Godot as additional data packs. That approach is preferred for DLCs, as it makes interacting with additional data packs seamless (virtual filesystem).

This ZIP archive support can be combined with runtime image, 3D scene and audio loading to provide a seamless modding experience without requiring users to go through the Godot editor to generate PCK/ZIP files.

Example that lists files in a ZIP archive in an ItemList node, then writes contents read from it to a new ZIP archive (essentially duplicating the archive):

```
# Load an existing ZIP archive.
var zip_reader = ZIPReader.new()
zip_reader.open(path)
var files = zip_reader.get_files()
# The list of files isn't sorted by default. Sort it for more
consistent processing.
files.sort()
for file in files:
    $ItemList.add_item(file, null)
    # Make folders disabled in the list.
    $ItemList.set_item_disabled(-1, file.ends_with("/"))

# Save a new ZIP archive.
var zip_packer = ZIPPacker.new()
var error = zip_packer.open(path)
if error != OK:
    push_error("Couldn't open path for saving ZIP archive
(error code: %s)." % error_string(error))
    return

# Reuse the above ZIPReader instance to read files from an
existing ZIP archive.
for file in zip_reader.get_files():
    zip_packer.start_file(file)
    zip_packer.write_file(zip_reader.read_file(file))
    zip_packer.close_file()

zip_packer.close()
```

Binary serialization API

Introduction

Godot has a serialization API based on Variant. It's used for converting data types to an array of bytes efficiently. This API is exposed via the global bytes_to_var() and var_to_bytes() functions, but it is also used in the `get_var` and `store_var` methods of FileAccess as well as the packet APIs for PacketPeer. This format is *not* used for binary scenes and resources.

Full Objects vs Object instance IDs

If a variable is serialized with `full_objects = true`, then any Objects contained in the variable will be serialized and included in the result. This is recursive.

If `full_objects = false`, then only the instance IDs will be serialized for any Objects contained in the variable.

Packet specification

The packet is designed to be always padded to 4 bytes. All values are little-endian-encoded. All packets have a 4-byte header representing an integer, specifying the type of data.

The lowest value two bytes are used to determine the type, while the highest value two bytes contain flags:

```
base_type = val & 0xFFFF;
```

```
flags = val >> 16;
```

Type	Value
0	null
1	bool
2	integer
3	float
4	string
5	vector2
6	rect2
7	vector3
8	transform2d
9	plane
10	quaternion

Type	Value
11	aabb
12	basis
13	transform3d
14	color
15	node path
16	rid
17	object
18	dictionary
19	array
20	raw array
21	int32 array
22	int64 array

Type	Value
23	float32 array
24	float64 array
25	string array
26	vector2 array
27	vector3 array
28	color array
29	max

Following this is the actual packet contents, which varies for each type of packet. Note that this assumes Godot is compiled with single-precision floats, which is the default. If Godot was compiled with double-precision floats, the length of "Float" fields within data structures should be 8, and the offset should be `(offset - 4) * 2 + 4`. The "float" type itself always uses double precision.

0: null

1: bool

Offset	Len	Type	Description
4	4	Integer	0 for False, 1 for True

2: int

If no flags are set (flags == 0), the integer is sent as a 32 bit integer:

Offset	Len	Type	Description
4	4	Integer	32-bit signed integer

If flag `ENCODE_FLAG_64` is set (`flags & 1 == 1`), the integer is sent as a 64-bit integer:

Offset	Len	Type	Description
4	8	Integer	64-bit signed integer

3: float

If no flags are set (flags == 0), the float is sent as a 32 bit single precision:

Offset	Len	Type	Description
4	4	Float	IEEE 754 single-precision float

If flag `ENCODE_FLAG_64` is set (`flags & 1 == 1`), the float is sent as a 64-bit double precision number:

Offset	Len	Type	Description
4	8	Float	IEEE 754 double-precision float

4: String

Offset	Len	Type	Description
4	4	Integer	String length (in bytes)
8	X	Bytes	UTF-8 encoded string

This field is padded to 4 bytes.

5: Vector2

Offset	Len	Type	Description
4	4	Float	X coordinate
8	4	Float	Y coordinate

6: Rect2

Offset	Len	Type	Description
4	4	Float	X coordinate
8	4	Float	Y coordinate
12	4	Float	X size
16	4	Float	Y size

7: Vector3

Offset	Len	Type	Description
4	4	Float	X coordinate

Offset	Len	Type	Description
8	4	Float	Y coordinate
12	4	Float	Z coordinate

8: Transform2D

Offset	Len	Type	Description
4	4	Float	The X component of the X column vector, accessed via [0][0]
8	4	Float	The Y component of the X column vector, accessed via [0][1]
12	4	Float	The X component of the Y column vector, accessed via [1][0]
16	4	Float	The Y component of the Y column vector, accessed via [1][1]
20	4	Float	The X component of the origin vector, accessed via [2][0]

Offset	Len	Type	Description
24	4	Float	The Y component of the origin vector, accessed via [2][1]

9: Plane

Offset	Len	Type	Description
4	4	Float	Normal X
8	4	Float	Normal Y
12	4	Float	Normal Z
16	4	Float	Distance

10: Quaternion

Offset	Len	Type	Description
4	4	Float	Imaginary X
8	4	Float	Imaginary Y

Offset	Len	Type	Description
12	4	Float	Imaginary Z
16	4	Float	Real W

11: AABB

Offset	Len	Type	Description
4	4	Float	X coordinate
8	4	Float	Y coordinate
12	4	Float	Z coordinate
16	4	Float	X size
20	4	Float	Y size
24	4	Float	Z size

12: Basis

Offset	Len	Type	Description
4	4	Float	The X component of the X column vector, accessed via [0][0]
8	4	Float	The Y component of the X column vector, accessed via [0][1]
12	4	Float	The Z component of the X column vector, accessed via [0][2]
16	4	Float	The X component of the Y column vector, accessed via [1][0]
20	4	Float	The Y component of the Y column vector, accessed via [1][1]
24	4	Float	The Z component of the Y column vector, accessed via [1][2]
28	4	Float	The X component of the Z column vector, accessed via [2][0]
32	4	Float	The Y component of the Z column vector, accessed via [2][1]
36	4	Float	The Z component of the Z column vector, accessed via [2][2]

Offset	Len	Type	Description

13: Transform3D

Offset	Len	Type	Description
4	4	Float	The X component of the X column vector, accessed via [0][0]
8	4	Float	The Y component of the X column vector, accessed via [0][1]
12	4	Float	The Z component of the X column vector, accessed via [0][2]
16	4	Float	The X component of the Y column vector, accessed via [1][0]
20	4	Float	The Y component of the Y column vector, accessed via [1][1]
24	4	Float	The Z component of the Y column vector, accessed via [1][2]
28	4	Float	The X component of the Z column vector, accessed via [2][0]

Offset	Len	Type	Description
32	4	Float	The Y component of the Z column vector, accessed via [2][1]
36	4	Float	The Z component of the Z column vector, accessed via [2][2]
40	4	Float	The X component of the origin vector, accessed via [3][0]
44	4	Float	The Y component of the origin vector, accessed via [3][1]
48	4	Float	The Z component of the origin vector, accessed via [3][2]

14: Color

Offset	Len	Type	Description
4	4	Float	Red (typically 0..1, can be above 1 for overbright colors)
8	4	Float	Green (typically 0..1, can be above 1 for overbright colors)

Offset	Len	Type	Description
12	4	Float	Blue (typically 0..1, can be above 1 for overbright colors)
16	4	Float	Alpha (0..1)

15: NodePath

Offset	Len	Type	Description
4	4	Integer	String length, or new format (val&0x80000000!=0 and NameCount=val&0x7FFFFFFF)

For old format:

Offset	Len	Type	Description
8	X	Bytes	UTF-8 encoded string

Padded to 4 bytes.

For new format:

Offset	**Len**	**Type**	**Description**
4	4	Integer	Sub-name count
8	4	Integer	Flags (absolute: val&1 != 0)

For each Name and Sub-Name

Offset	**Len**	**Type**	**Description**
X+0	4	Integer	String length
X+4	X	Bytes	UTF-8 encoded string

Every name string is padded to 4 bytes.

16: RID (unsupported)

17: Object

An Object could be serialized in three different ways: as a null value, with `full_objects = false`, or with `full_objects = true`.

A null value

Offset	Len	Type	Description
4	4	Integer	Zero (32-bit signed integer)

`full_objects` disabled

Offset	Len	Type	Description
4	8	Integer	The Object instance ID (64-bit signed integer)

`full_objects` enabled

Offset	Len	Type	Description
4	4	Integer	Class name (String length)
8	X	Bytes	Class name (UTF-8 encoded string)
X+8	4	Integer	The number of properties that are serialized

For each property:

Offset	Len	Type	Description
Y	4	Integer	Property name (String length)
Y+4	Z	Bytes	Property name (UTF-8 encoded string)
Y+4+Z	W	<variable>	Property value, using this same format

Note

Not all properties are included. Only properties that are configured with the PROPERTY_USAGE_STORAGE flag set will be serialized. You can add a new usage flag to a property by overriding the _get_property_list method in your class. You can also check how property usage is configured by calling `Object._get_property_list` See PropertyUsageFlags for the possible usage flags.

18: Dictionary

Offset	Len	Type	Description
4	4	Integer	val&0x7FFFFFFF = elements, val&0x80000000 = shared (bool)

Then what follows is, for amount of "elements", pairs of key and value, one after the other, using this same format.

19: Array

Offset	Len	Type	Description
4	4	Integer	val&0x7FFFFFFF = elements, val&0x80000000 = shared (bool)

Then what follows is, for amount of "elements", values one after the other, using this same format.

20: PackedByteArray

Offset	Len	Type	Description
4	4	Integer	Array length (Bytes)
8..8+length	1	Byte	Byte (0..255)

The array data is padded to 4 bytes.

21: PackedInt32Array

Offset	Len	Type	Description
4	4	Integer	Array length (Integers)

Offset	Len	Type	Description
8..8+length*4	4	Integer	32-bit signed integer

22: PackedInt64Array

Offset	Len	Type	Description
4	8	Integer	Array length (Integers)
8..8+length*8	8	Integer	64-bit signed integer

23: PackedFloat32Array

Offset	Len	Type	Description
4	4	Integer	Array length (Floats)
8..8+length*4	4	Integer	32-bit IEEE 754 single-precision float

24: PackedFloat64Array

Offset	Len	Type	Description
4	4	Integer	Array length (Floats)
8..8+length*8	8	Integer	64-bit IEEE 754 double-precision float

25: PackedStringArray

Offset	Len	Type	Description
4	4	Integer	Array length (Strings)

For each String:

Offset	Len	Type	Description
X+0	4	Integer	String length
X+4	X	Bytes	UTF-8 encoded string

Every string is padded to 4 bytes.

26: PackedVector2Array

Offset	Len	Type	Description
4	4	Integer	Array length
8..8+length*8	4	Float	X coordinate
8..12+length*8	4	Float	Y coordinate

27: PackedVector3Array

Offset	Len	Type	Description
4	4	Integer	Array length
8..8+length*12	4	Float	X coordinate
8..12+length*12	4	Float	Y coordinate
8..16+length*12	4	Float	Z coordinate

28: PackedColorArray

Offset	Len	Type	Description
4	4	Integer	Array length
8..8+length*16	4	Float	Red (typically 0..1, can be above 1 for overbright colors)
8..12+length*16	4	Float	Green (typically 0..1, can be above 1 for overbright colors)
8..16+length*16	4	Float	Blue (typically 0..1, can be above 1 for overbright colors)
8..20+length*16	4	Float	Alpha (0..1)

Internationalization

- Internationalizing games
- Localization using gettext
- Locale codes
- Pseudolocalization

Internationalizing games

Introduction

While indie or niche games usually do not need localization, games targeting a more massive market often require localization. Godot offers many tools to make this process more straightforward, so this tutorial is more like a collection of tips and tricks.

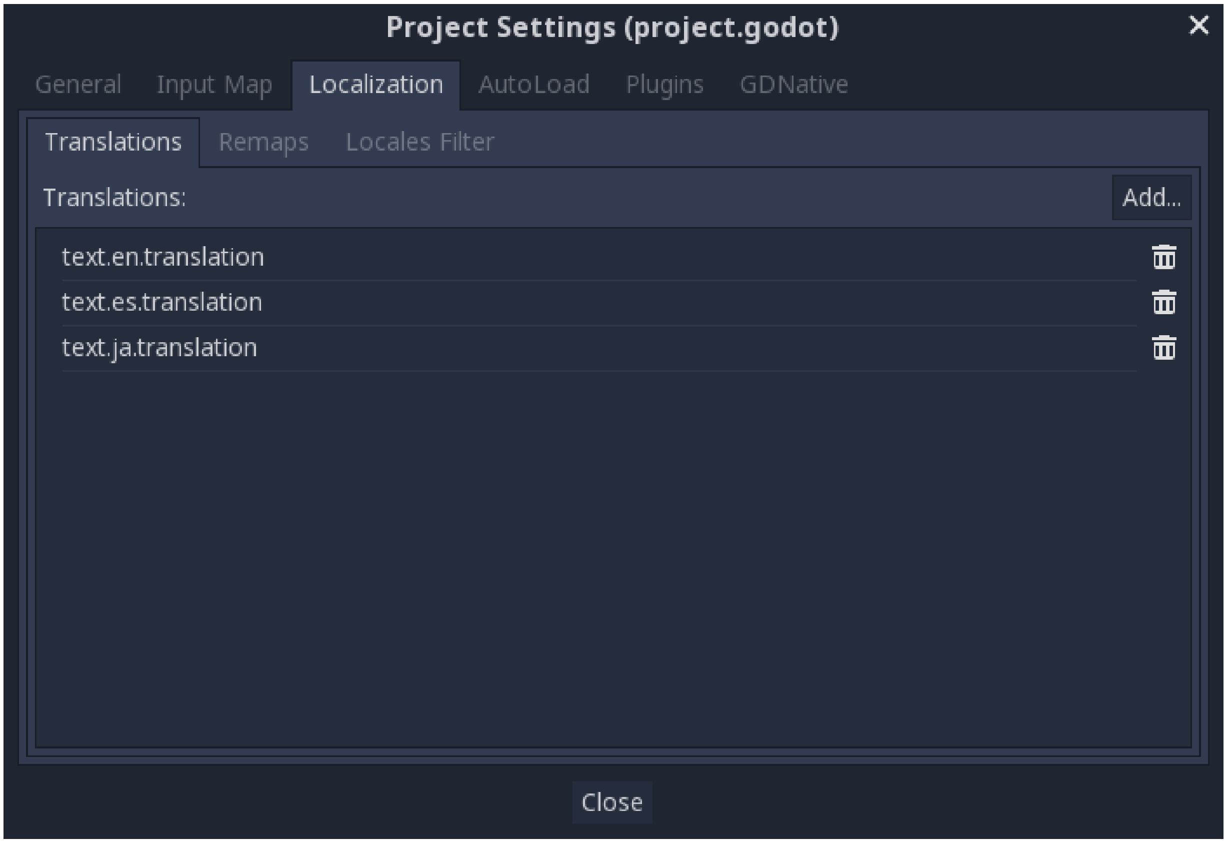
Localization is usually done by specific studios hired for the job. Despite the huge amount of software and file formats available for this, the most common way to do localization to this day is still with spreadsheets. The process of creating the spreadsheets and importing them is already covered in the Importing translations tutorial. If you haven't read the Importing translations page before, we recommend you give it a read before reading this page.

Note

We will be using the official demo as an example; you can download it from the Asset Library [https://godotengine.org/asset-library/asset/2776].

Configuring the imported translation

Translations can get updated and re-imported when they change, but they still have to be added to the project. This is done in **Project** → **Project Settings** → **Localization**:


Project Settings (project.godot)
General
Input Map
Localization
AutoLoad
Plugins
GDNative
Translations
Remaps
Locales Filter
Translations:
Add...
Close

text.en.translation

text.es.translation

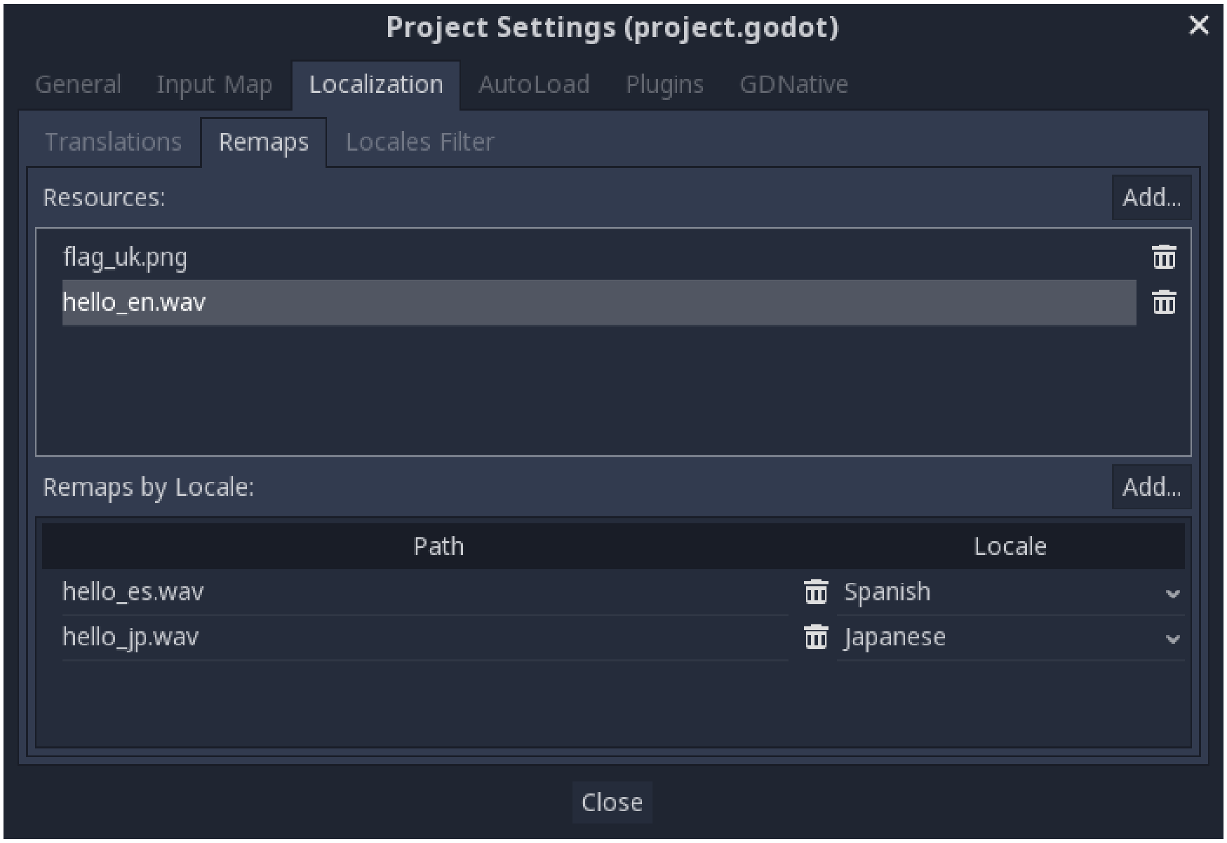
text.ja.translation

The above dialog is used to add or remove translations project-wide.

Localizing resources

It is also possible to instruct Godot to use alternate versions of assets (resources) depending on the current language. This can be used for localized images such as in-game billboards or localized voices.

The **Remaps** tab can be used for this:


Project Settings (project.godot)
General
Input Map
Localization
AutoLoad
Plugins
GDNative
Translations
Remaps
Locales Filter
Add...
Add...
Close

Resources:

flag_uk.png
hello_en.wav

Remaps by Locale:

Path	Locale
hello_es.wav	Spanish
hello_jp.wav	Japanese

Select the resource to be remapped then add some alternatives for each locale.

Note

The resource remapping system isn't supported for DynamicFonts. To use different fonts depending on the language's script, use the DynamicFont fallback system instead, which lets you define as many fallback fonts as you want.

The upside of the DynamicFont fallback system is that it works regardless of the current language, making it ideal for things like multiplayer chat where the text language may not match the client's language.

Converting keys to text

Some controls, such as Button and Label, will automatically fetch a translation if their text matches a translation key. For example, if a label's text is "MAIN_SCREEN_GREETING1" and that key exists in the current translation, then the text will automatically be translated.

This automatic translation behavior may be undesirable in certain cases. For instance, when using a Label to display a player's name, you most likely don't want the player's name to be translated if it matches a translation key. To disable automatic translation on a specific node, disable **Localization > Auto Translate** in the inspector.

In code, the Object.tr() function can be used. This will just look up the text in the translations and convert it if found:

```
level.text = tr("LEVEL_5_NAME")
status.text = tr("GAME_STATUS_%d" % status_index)
```

Note

If no text is displayed after changing the language, try to use a different font. The default project font only supports a subset of the Latin-1 character set, which cannot be used to display languages like Russian or Chinese.

A good resource for multilingual fonts is Noto Fonts [https://www.google.com/get/noto/]. Make sure to download the correct variation if you're using a less common language.

Once you've downloaded the font, load the TTF file into a DynamicFont resource and use it as a custom font of your Control node. For better reusability, associate a new a Theme resource to your root Control node and define the DynamicFont as the Default Font in the theme.

Placeholders

To feature placeholders in your translated strings, use GDScript format strings or the equivalent feature in C#. This lets translators move the location of the placeholder in the string freely, which allows translations to sound more natural. Named placeholders with the `String.format()` function should be used whenever possible, as they also allow translators to choose the *order* in which placeholders appear:

```
# The placeholder's locations can be changed, but not their
order.
# This will probably not suffice for some target languages.
message.text = tr("%s picked up the %s") % ["Ogre", "Sword"]

# The placeholder's locations and order can be changed.
# Additionally, this form gives more context for translators to
work with.
message.text = tr("{character} picked up the
{weapon}").format({character = "Ogre", weapon = "Sword"})
```

Translation contexts

If you're using plain English as source strings (rather than message codes `LIKE_THIS`), you may run into ambiguities when you have to translate the same English string to different strings in certain target languages. You can optionally specify a *translation context* to resolve this ambiguity and allow target languages to use different strings, even though the source string is identical:

```
# "Close", as in an action (to close something).
button.set_text(tr("Close", "Actions"))
```

```
# "Close", as in a distance (opposite of "far").
distance_label.set_text(tr("Close", "Distance"))
```

Pluralization

Most languages require different strings depending on whether an object is in singular or plural form. However, hardcoding the "is plural" condition depending on whether there is more than 1 object is not valid in all languages.

Some languages have more than two plural forms, and the rules on the number of objects required for each plural form vary. Godot offers support for *pluralization* so that the target locales can handle this automatically.

Pluralization is meant to be used with positive (or zero) integer numbers only. Negative and floating-point values usually represent physical entities for which singular and plural don't clearly apply.

```
var num_apples = 5
label.text = tr_n("There is %d apple", "There are %d apples",
num_apples) % num_apples
```

This can be combined with a context if needed:

```
var num_jobs = 1
label.text = tr_n("%d job", "%d jobs", num_jobs, "Task
Manager") % num_jobs
```

Note

Providing pluralized translations is only supported with Localization using gettext, not CSV.

Making controls resizable

The same text in different languages can vary greatly in length. For this, make sure to read the tutorial on Size and anchors, as dynamically adjusting control sizes may help. Container can be useful, as well as the text wrapping options available in Label.

To check whether your UI can accommodate translations with longer strings than the original, you can enable *pseudolocalization* in the advanced Project Settings. This will replace all your localizable strings with longer versions of themselves, while also replacing some characters in the original strings with accented versions (while still being readable). Placeholders are kept as-is, so that they keep working when pseudolocalization is enabled.

For example, the string `Hello world, this is %s!` becomes `[Ĥéłłô ŵôŕłd́, ŧh̀íš íš %s!]` when pseudolocalization is enabled.

While looking strange at first, pseudolocalization has several benefits:

- It lets you spot non-localizable strings quickly, so you can go over them and make them localizable (if it makes sense to do so).
- It lets you check UI elements that can't fit long strings. Many languages will feature much longer translations than the source text, so it's important to ensure your UI can accommodate longer-than-usual strings.
- It lets you check whether your font contains all the characters required to support various languages. However, since the goal of pseudolocalization is to keep the original strings readable, it's not an effective test for checking whether a font can support CJK or right-to-left languages.

The project settings allow you to tune pseudolocalization behavior, so that you can disable parts of it if desired.

TranslationServer

Godot has a server handling low-level translation management called the TranslationServer. Translations can be added or removed during run-time; the current language can also be changed at run-time.

Bidirectional text and UI Mirroring

Arabic and Hebrew are written from right to left (except for the numbers and Latin words mixed in), and the user interface for these languages should be mirrored as well. In some languages the shape of a glyph changes depending on the surrounding characters.

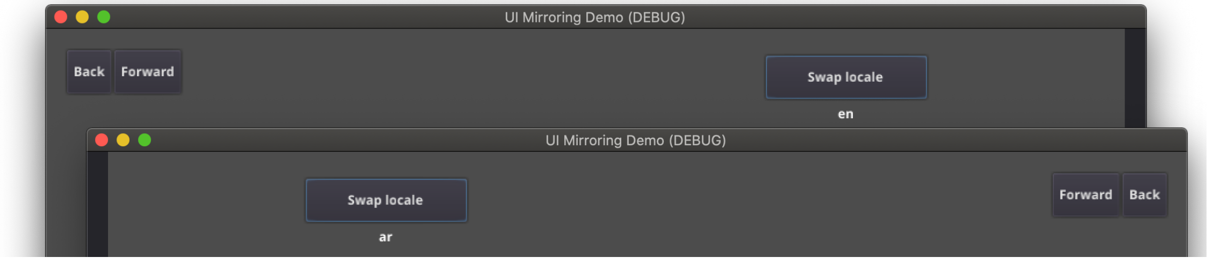
Support for bidirectional writing systems and UI mirroring is transparent, you don't usually need to change anything or have any knowledge of the specific writing system.

For RTL languages, Godot will automatically do the following changes to the UI:

- Mirrors left/right anchors and margins.
- Swaps left and right text alignment.
- Mirrors horizontal order of the child controls in the containers, and items in Tree/ItemList controls.
- Uses mirrored order of the internal control elements (e.g. OptionButton dropdown button, checkbox alignment, List column order, Tree item icons and connecting line alignment, e.t.c.), in some cases mirrored controls use separate theme styles.
- Coordinate system is not mirrored, and non-UI nodes (sprites, e.t.c) are not affected.

It is possible to override text and control layout direction by using the following control properties:

- `text_direction`, sets the base text direction. When set to "auto", direction depends on the first strong directional character in the text according to the Unicode Bidirectional Algorithm,
- `language`, overrides current project locale.
- `structured_text_bidi_override` property and `_structured_text_parser` callback, enables special handling for structured text.
- `layout_direction`, overrides control mirroring.



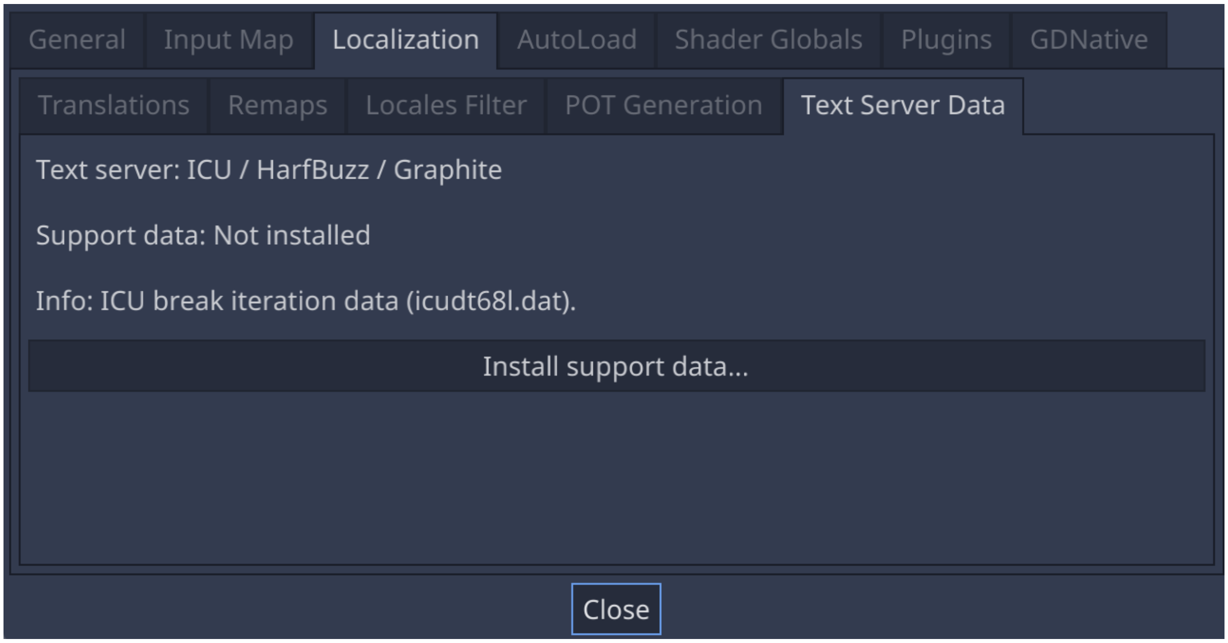
See also

You can see how right-to-left typesetting works in action using the BiDI and Font Features demo project [https://github.com/godotengine/godot-demo-projects/tree/master/gui/bidi_and_font_features].

Adding break iterator data to exported project

Some languages are written without spaces, and word and line breaking requires more than rules over character sequences. Godot includes ICU rule and dictionary based, break iterator data, but this data is not included into exported projects by default. To include it go to **Project** → **Project Settings** → **Localization** → **Text**

Server Data and click **Install support data...**. Break iterator data is about 4 MB large.


General
Input Map
Localization
AutoLoad
Shader Globals
Plugins
GDNative
Translations
Remaps
Locales Filter
POT Generation
Text Server Data
Install support data...
Close

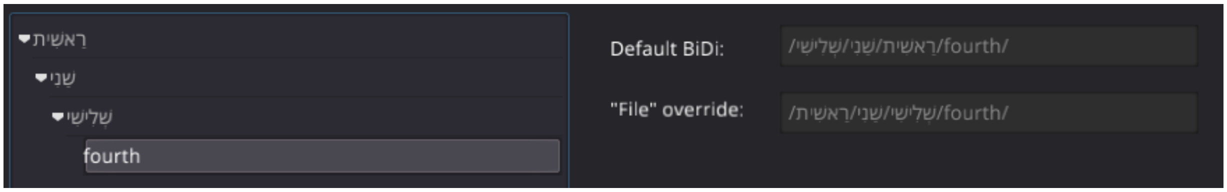
Text server: ICU / HarfBuzz / Graphite

Support data: Not installed

Info: ICU break iteration data (icudt68l.dat).

Structured text BiDi override

Unicode BiDi algorithm is designed to work with natural text and it's incapable of handling text with the higher level order, like file names, URIs, email addresses, regular expressions or source code.


רַאשִׁית
שֵׁנִי
שְׁלִישִׁי
fourth

Default BiDi: /שְׁלִישִׁי/שֵׁנִי/רַאשִׁית/fourth/

"File" override: /רַאשִׁית/שֵׁנִי/שְׁלִישִׁי/fourth/

For example, the path for this shown directory structure will be displayed incorrectly (top "LineEdit" control). "File" type structured text override splits text into segments, then BiDi algorithm is applied to each of them individually to correctly display directory names in any language and preserve correct order of the folders (bottom "LineEdit" control).

Custom callbacks provide a way to override BiDi for the other types of structured text.

Localizing numbers

Controls specifically designed for number input or output (e.g. ProgressBar, SpinBox) will use localized numbering system automatically, for the other control TextServer.format_number(string, language) can be used to convert Western Arabic numbers (0..9) to the localized numbering system and TextServer.parse_number(string, language) to convert it back.

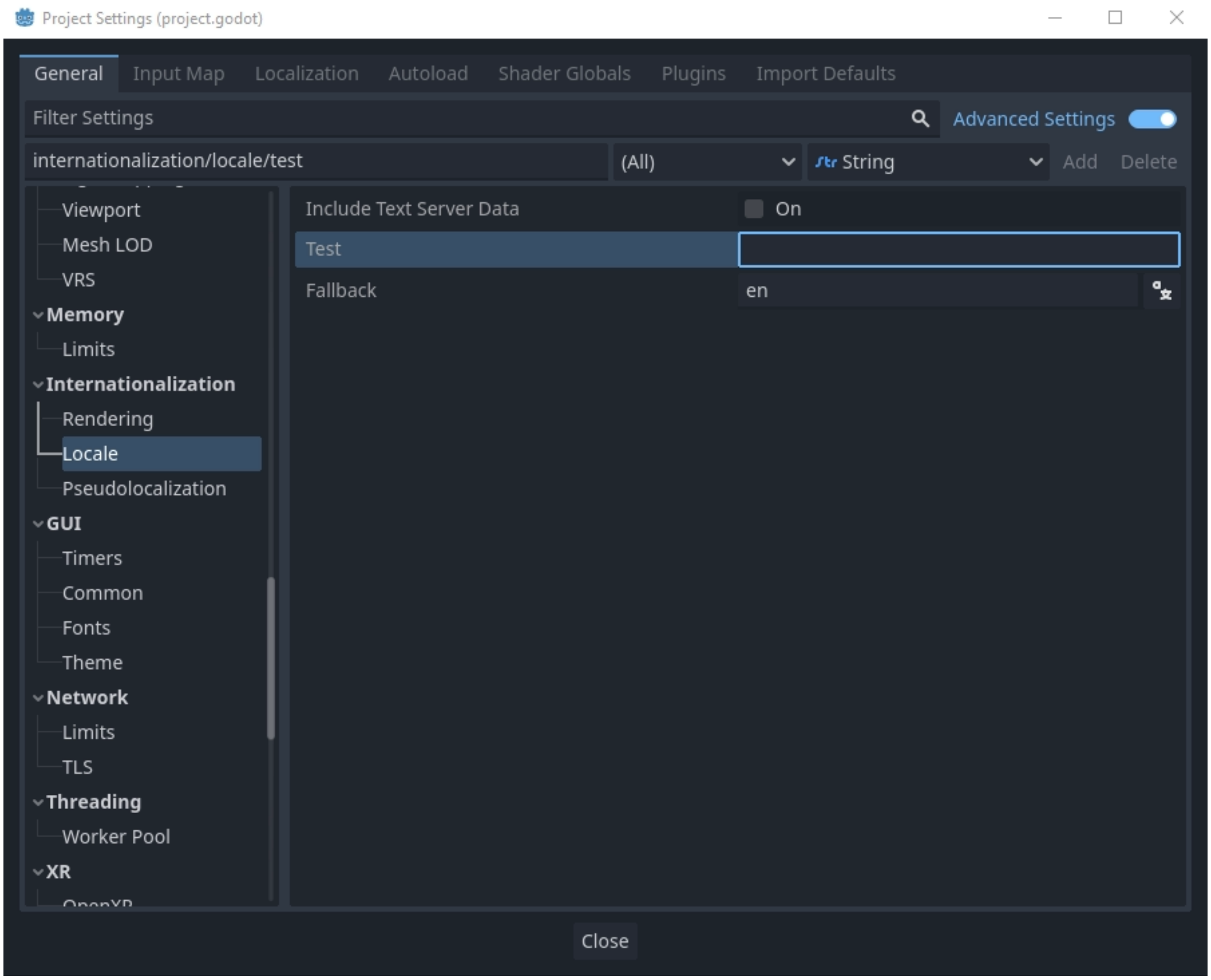
Localizing icons and images

Icons with left and right pointing arrows which may need to be reversed for Arabic and Hebrew locales, in case they indicate movement or direction (e.g. back/forward buttons). Otherwise, they can remain the same.

Testing translations

You may want to test a project's translation before releasing it. Godot provides two ways to do this.

First, in the Project Settings, under **Internationalization > Locale** (with advanced settings enabled), there is a **Test** property. Set this property to the locale code of the language you want to test. Godot will run the project with that locale when the project is run (either from the editor or when exported).


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Filter Settings
Advanced Settings
(All)
String
Add
Delete
Viewport
Mesh LOD
VRS
Memory
Limits
Internationalization
Rendering
Locale
Pseudolocalization
GUI
Timers
Common
Fonts
Theme
Network
Limits
TLS
Threading
Worker Pool
XR
OpenXR
Close

internationalization/locale/test

Include Text Server Data ☐ On

Test

Fallback en

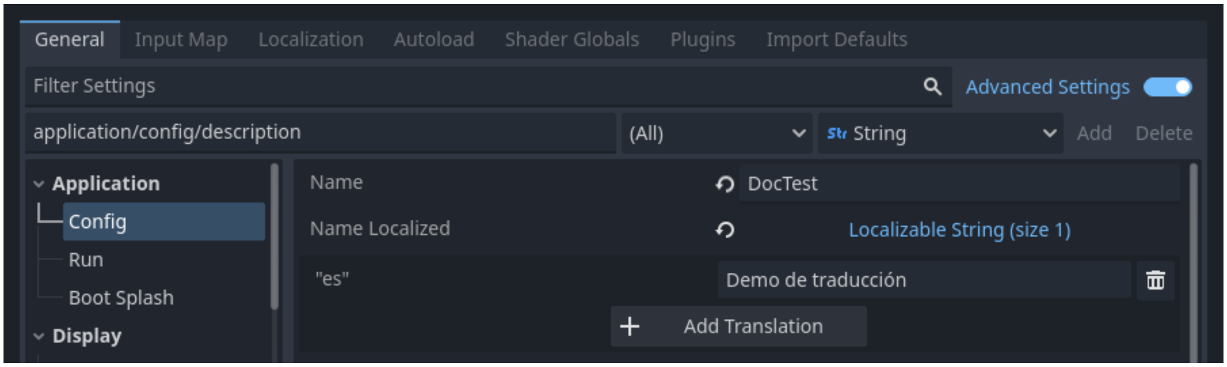
Keep in mind that since this is a project setting, it will show up in version control when it is set to a non-empty value. Therefore, it should be set back to an empty value before committing changes to version control.

Translations can also be tested when running Godot from the command line. For example, to test a game in French, the following argument can be supplied:

```
godot --language fr
```

Translating the project name

The project name becomes the app name when exporting to different operating systems and platforms. To specify the project name in more than one language go to **Project > Project Settings> Application > Config**. From here click on the button that says `Localizable String (Size 0)`. Now there should be a button below that which says `Add Translation`. Click on that and it will take you to a page where you can choose the language (and country if needed) for your project name translation. After doing that you can now type in the localized name.


General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Filter Settings
Advanced Settings
application/config/description
(All)
String
Add
Delete
Application
Config
Run
Boot Splash
Display
Add Translation

Name: DocTest
Name Localized: Localizable String (size 1)
"es": Demo de traducción

If you are unsure about the language code to use, refer to the list of locale codes.

Localization using gettext

In addition to Importing translations in CSV format, Godot also supports loading translation files written in the GNU gettext format (text-based `.po` and compiled `.mo` since Godot 4.0).

Note

For an introduction to gettext, check out A Quick Gettext Tutorial [https://www.labri.fr/perso/fleury/posts/programming/a-quick-gettext-tutorial.html]. It's written with C projects in mind, but much of the advice also applies to Godot (with the exception of `xgettext`).

Advantages

- gettext is a standard format, which can be edited using any text editor or GUI editors such as Poedit [https://poedit.net/].
- gettext is supported by translation platforms such as Transifex [https://www.transifex.com/] and Weblate [https://weblate.org/], which makes it easier for people to collaborate to localization.
- Compared to CSV, gettext works better with version control systems like Git, as each locale has its own messages file.
- Multiline strings are more convenient to edit in gettext files compared to CSV files.

Disadvantages

- gettext is a more complex format than CSV and can be harder to grasp for people new to software localization.

- People who maintain localization files will have to install gettext tools on their system. However, as Godot supports using text-based message files (`.po`), translators can test their work without having to install gettext tools.

Installing gettext tools

The command line gettext tools are required to perform maintenance operations, such as updating message files. Therefore, it's strongly recommended to install them.

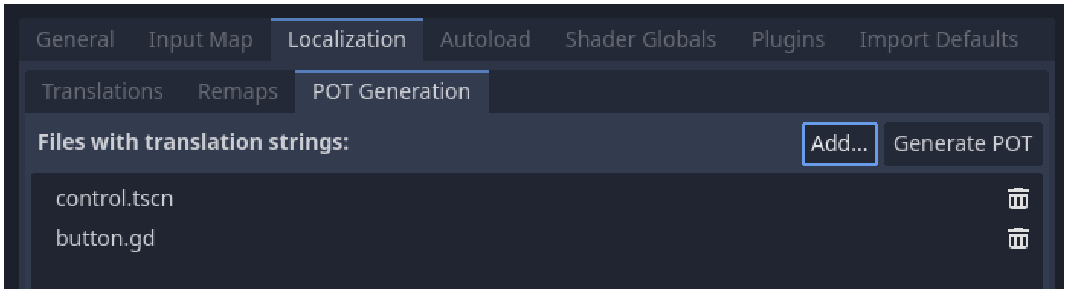
- **Windows:** Download an installer from this page [https://mlocati.github.io/articles/gettext-iconv-windows.html]. Any architecture and binary type (shared or static) works; if in doubt, choose the 64-bit static installer.
- **macOS:** Install gettext either using Homebrew [https://brew.sh/] with the `brew install gettext` command, or using MacPorts [https://www.macports.org/] with the `sudo port install gettext` command.
- **Linux:** On most distributions, install the `gettext` package from your distribution's package manager.

Creating the PO template

Automatic generation using the editor

Since Godot 4.0, the editor can generate a PO template automatically from specified scene and GDScript files. This POT generation also supports translation contexts and pluralization if used in a script, with the optional second argument of `tr()` and the `tr_n()` method.

Open the Project Settings' **Localization > POT Generation** tab, then use the **Add...** button to specify the path to your project's scenes and scripts that contain localizable strings:


General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Translations
Remaps
POT Generation
Files with translation strings:
Add...
Generate POT
control.tscn
button.gd

Creating a PO template in the **Localization > POT Generation** tab of the Project Settings

After adding at least one scene or script, click **Generate POT** in the top-right corner, then specify the path to the output file. This file can be placed anywhere in the project directory, but it's recommended to keep it in a subdirectory such as `locale`, as each locale will be defined in its own file.

You can then move over to [creating a messages file from a PO template](#).

Note

Remember to regenerate the PO template after making any changes to localizable strings, or after adding new scenes or scripts. Otherwise, newly added strings will not be localizable and translators won't be able to update translations for outdated strings.

Manual creation

If the automatic generation approach doesn't work out for your needs, you can create a PO template by hand in a text editor. This file can be placed anywhere in the project directory, but it's recommended to keep it in a subdirectory, as each locale will be defined in its own file.

Create a directory named `locale` in the project directory. In this directory, save a file named `messages.pot` with the following contents:

```
# Don't remove the two lines below, they're required for
gettext to work correctly.
msgid ""
msgstr ""

# Example of a regular string.
msgid "Hello world!"
msgstr ""

# Example of a string with pluralization.
msgid "There is %d apple."
msgid_plural "There are %d apples."
msgstr[0] ""
msgstr[1] ""

# Example of a string with a translation context.
msgctxt "Actions"
msgid "Close"
msgstr ""
```

Messages in gettext are made of `msgid` and `msgstr` pairs. `msgid` is the source string (usually in English), `msgstr` will be the translated string.

Warning

The `msgstr` value in PO template files (`.pot`) should **always** be empty. Localization will be done in the generated `.po` files instead.

Creating a messages file from a PO template

The `msginit` command is used to turn a PO template into a messages file. For instance, to create a French localization file, use the following command while in the `locale` directory:

```
msginit --no-translator --input=messages.pot --locale=fr
```

The command above will create a file named `fr.po` in the same directory as the PO template.

Alternatively, you can do that graphically using Poedit, or by uploading the POT file to your web platform of choice.

Loading a messages file in Godot

To register a messages file as a translation in a project, open the **Project Settings**, then go to the **Localization** tab. In **Translations**, click **Add...** then choose the `.po` or `.mo` file in the file dialog. The locale will be inferred from the `"Language: <code>\n"` property in the messages file.

Note

See Internationalizing games for more information on importing and testing translations in Godot.

Updating message files to follow the PO template

After updating the PO template, you will have to update message files so that they contain new strings, while removing strings that are no longer present in the PO template. This can be done automatically using the `msgmerge` tool:

```
# The order matters: specify the message file *then* the PO template!
msgmerge --update --backup=none fr.po messages.pot
```

If you want to keep a backup of the original message file (which would be saved as `fr.po~` in this example), remove the `--backup=none` argument.

Note

After running `msgmerge`, strings which were modified in the source language will have a "fuzzy" comment added before them in the `.po` file. This comment denotes that the translation should be updated to match the new source string, as the translation will most likely be inaccurate until it's updated.

Strings with "fuzzy" comments will **not** be read by Godot until the translation is updated and the "fuzzy" comment is removed.

Checking the validity of a PO file or template

It is possible to check whether a gettext file's syntax is valid by running the command below:

```
msgfmt fr.po --check
```

If there are syntax errors or warnings, they will be displayed in the console. Otherwise, `msgfmt` won't output anything.

Using binary MO files (useful for large projects only)

For large projects with several thousands of strings to translate or more, it can be worth it to use binary (compiled) MO message files instead of text-based PO files. Binary MO files are smaller and faster to read than the equivalent PO files.

You can generate an MO file with the command below:

```
msgfmt fr.po --no-hash -o fr.mo
```

If the PO file is valid, this command will create a `fr.mo` file besides the PO file. This MO file can then be loaded in Godot as described above.

The original PO file should be kept in version control so you can update your translation in the future. In case you lose the original PO file and wish to decompile an MO file into a text-based PO file, you can do so with:

```
msgunfmt fr.mo > fr.po
```

The decompiled file will not include comments or fuzzy strings, as these are never compiled in the MO file in the first place.

Locale codes

Locale code has the following format: `language_Script_COUNTRY_VARIANT`, where:

- `language` - 2 or 3-letter language code, in lower case.
- `Script` - optional, 4-letter script code, in title case.
- `COUNTRY` - optional, 2-letter country code, in upper case.
- `VARIANT` - optional, language variant, region and, sort order. A variant can have any number of underscored keywords.

List of supported language codes

Language code	Name
aa	Afar
ab	Abkhazian
ace	Achinese
ach	Acoli
ada	Adangme
ady	Adyghe

Language code	Name
ae	Avestan
aeb	Tunisian Arabic
af	Afrikaans
afh	Afrihili
agq	Aghem
ain	Ainu
agr	Aguaruna
ak	Akan
akk	Akkadian
akz	Alabama
ale	Aleut
aln	Gheg Albanian

Language code	Name
alt	Southern Altai
am	Amharic
an	Aragonese
ang	Old English
anp	Angika
ar	Arabic
arc	Aramaic
arn	Mapudungun
aro	Araona
arp	Arapaho
arq	Algerian Arabic
ars	Najdi Arabic

Language code	Name
arw	Arawak
ary	Moroccan Arabic
arz	Egyptian Arabic
as	Assamese
asa	Asu
ase	American Sign Language
ast	Asturian
av	Avaric
avk	Kotava
awa	Awadhi
ayc	Southern Aymara
ay	Aymara

Language code	Name
az	Azerbaijani
ba	Bashkir
bal	Baluchi
ban	Balinese
bar	Bavarian
bas	Bassa
bax	Bamun
bbc	Batak Toba
bbj	Ghomala
be	Belarusian
bej	Beja
bem	Bemba

Language code	Name
ber	Berber
bew	Betawi
bez	Bena
bfd	Bafut
bfq	Badaga
bg	Bulgarian
bhb	Bhili
bgn	Western Balochi
bho	Bhojpuri
bi	Bislama
bik	Bikol

Language code	Name
bin	Bini
bjn	Banjar
bkm	Kom
bla	Siksika
bm	Bambara
bn	Bengali
bo	Tibetan
bpy	Bishnupriya
bqi	Bakhtiari
br	Breton
brh	Brahui
brx	Bodo

Language code	Name
bs	Bosnian
bss	Akoose
bua	Buriat
bug	Buginese
bum	Bulu
byn	Bilin
byv	Medumba
ca	Catalan
cad	Caddo
car	Carib
cay	Cayuga
cch	Atsam

Language code	Name
ccp	Chakma
ce	Chechen
ceb	Cebuano
cgg	Chiga
ch	Chamorro
chb	Chibcha
chg	Chagatai
chk	Chuukese
chm	Mari
chn	Chinook Jargon
cho	Choctaw
chp	Chipewyan

Language code	Name
chr	Cherokee
chy	Cheyenne
cic	Chickasaw
ckb	Central Kurdish
csb	Kashubian
cmn	Mandarin Chinese
co	Corsican
cop	Coptic
cps	Capiznon
cr	Cree
crh	Crimean Tatar
crs	Seselwa Creole French

Language code	Name
cs	Czech
csb	Kashubian
cu	Church Slavic
cv	Chuvash
cy	Welsh
da	Danish
dak	Dakota
dar	Dargwa
dav	Taita
de	German
del	Delaware

Language code	Name
den	Slave
dgr	Dogrib
din	Dinka
dje	Zarma
doi	Dogri
dsb	Lower Sorbian
dtp	Central Dusun
dua	Duala
dum	Middle Dutch
dv	Dhivehi
dyo	Jola-Fonyi
dyu	Dyula

Language code	Name
dz	Dzongkha
dzg	Dazaga
ebu	Embu
ee	Ewe
efi	Efik
egl	Emilian
egy	Ancient Egyptian
eka	Ekajuk
el	Greek
elx	Elamite
en	English
enm	Middle English

Language code	Name
eo	Esperanto
es	Spanish
esu	Central Yupik
et	Estonian
eu	Basque
ewo	Ewondo
ext	Extremaduran
fa	Persian
fan	Fang
fat	Fanti
ff	Fulah
fi	Finnish

Language code	Name
fil	Filipino
fit	Tornedalen Finnish
fj	Fijian
fo	Faroese
fon	Fon
fr	French
frc	Cajun French
frm	Middle French
fro	Old French
frp	Arpitan
frr	Northern Frisian
frs	Eastern Frisian

Language code	Name
fur	Friulian
fy	Western Frisian
ga	Irish
gaa	Ga
gag	Gagauz
gan	Gan Chinese
gay	Gayo
gba	Gbaya
gbz	Zoroastrian Dari
gd	Scottish Gaelic
gez	Geez

Language code	Name
gil	Gilbertese
gl	Galician
glk	Gilaki
gmh	Middle High German
gn	Guarani
goh	Old High German
gom	Goan Konkani
gon	Gondi
gor	Gorontalo
got	Gothic
grb	Grebo
grc	Ancient Greek

Language code	Name
gsw	Swiss German
gu	Gujarati
guc	Wayuu
gur	Frafra
guz	Gusii
gv	Manx
gwi	Gwich’in
ha	Hausa
hai	Haida
hak	Hakka Chinese
haw	Hawaiian
he, iw	Hebrew

Language code	Name
hi	Hindi
hif	Fiji Hindi
hil	Hiligaynon
hit	Hittite
hmn	Hmong
ho	Hiri Motu
hne	Chhattisgarhi
hr	Croatian
hsb	Upper Sorbian
hsn	Xiang Chinese
ht	Haitian
hu	Hungarian

Language code	Name
hup	Hupa
hus	Huastec
hy	Armenian
hz	Herero
ia	Interlingua
iba	Iban
ibb	Ibibio
id, in	Indonesian
ie	Interlingue
ig	Igbo
ii	Sichuan Yi
ik	Inupiaq

Language code	Name
ilo	Iloko
inh	Ingush
io	Ido
is	Icelandic
it	Italian
iu	Inuktitut
izh	Ingrian
ja	Japanese
jam	Jamaican Creole English
jbo	Lojban
jgo	Ngomba

Language code	Name
jmc	Machame
jpr	Judeo-Persian
jrb	Judeo-Arabic
jut	Jutish
jv	Javanese
ka	Georgian
kaa	Kara-Kalpak
kab	Kabyle
kac	Kachin
kaj	Jju
kam	Kamba
kaw	Kawi

Language code	Name
kbd	Kabardian
kbl	Kanembu
kcg	Tyap
kde	Makonde
kea	Kabuverdianu
ken	Kenyang
kfo	Koro
kg	Kongo
kgp	Kaingang
kha	Khasi
kho	Khotanese
khq	Koyra Chiini

Language code	Name
khw	Khowar
ki	Kikuyu
kiu	Kirmanjki
kj	Kuanyama
kk	Kazakh
kkj	Kako
kl	Kalaallisut
kln	Kalenjin
km	Central Khmer
kmb	Kimbundu
kn	Kannada
ko	Korean

Language code	Name
koi	Komi-Permyak
kok	Konkani
kos	Kosraean
kpe	Kpelle
kr	Kanuri
krc	Karachay-Balkar
kri	Krio
krj	Kinaray-a
krl	Karelian
kru	Kurukh
ks	Kashmiri
ksb	Shambala

Language code	Name
ksf	Bafia
ksh	Colognian
ku	Kurdish
kum	Kumyk
kut	Kutenai
kv	Komi
kw	Cornish
ky	Kirghiz
lag	Langi
la	Latin
lad	Ladino

Language code	Name
lag	Langi
lah	Lahnda
lam	Lamba
lb	Luxembourgish
lez	Lezghian
lfn	Lingua Franca Nova
lg	Ganda
li	Limburgan
lij	Ligurian
liv	Livonian
lkt	Lakota
lmo	Lombard

Language code	Name
ln	Lingala
lo	Lao
lol	Mongo
lou	Louisiana Creole
loz	Lozi
lrc	Northern Luri
lt	Lithuanian
ltg	Latgalian
lu	Luba-Katanga
lua	Luba-Lulua
lui	Luiseno
lun	Lunda

Language code	Name
luo	Luo
lus	Mizo
luy	Luyia
lv	Latvian
lzh	Literary Chinese
lzz	Laz
mad	Madurese
maf	Mafa
mag	Magahi
mai	Maithili
mak	Makasar
man	Mandingo

Language code	Name
mas	Masai
mde	Maba
mdf	Moksha
mdr	Mandar
men	Mende
mer	Meru
mfe	Morisyen
mg	Malagasy
mga	Middle Irish
mgh	Makhuwa-Meetto
mgo	Meta’
mh	Marshallese

Language code	Name
mhr	Eastern Mari
mi	Māori
mic	Mi'kmaq
min	Minangkabau
miq	Mískito
mjw	Karbi
mk	Macedonian
ml	Malayalam
mn	Mongolian
mnc	Manchu
mni	Manipuri

Language code	**Name**
mnw	Mon
mos	Mossi
moh	Mohawk
mr	Marathi
mrj	Western Mari
ms	Malay
mt	Maltese
mua	Mundang
mus	Muscogee
mwl	Mirandese
mwr	Marwari
mwv	Mentawai

Language code	Name
my	Burmese
mye	Myene
myv	Erzya
mzn	Mazanderani
na	Nauru
nah	Nahuatl
nan	Min Nan Chinese
nap	Neapolitan
naq	Nama
nan	Min Nan Chinese
nb, no	Norwegian Bokmål
nd	North Ndebele

Language code	Name
nds	Low German
ne	Nepali
new	Newari
nhn	Central Nahuatl
ng	Ndonga
nia	Nias
niu	Niuean
njo	Ao Naga
nl	Dutch
nmg	Kwasio
nn	Norwegian Nynorsk
nnh	Ngiemboon

Language code	Name
nog	Nogai
non	Old Norse
nov	Novial
nqo	N'ko
nr	South Ndebele
nso	Pedi
nus	Nuer
nv	Navajo
nwc	Classical Newari
ny	Nyanja
nym	Nyamwezi
nyn	Nyankole

Language code	Name
nyo	Nyoro
nzi	Nzima
oc	Occitan
oj	Ojibwa
om	Oromo
or	Odia
os	Ossetic
osa	Osage
ota	Ottoman Turkish
pa	Panjabi
pag	Pangasinan

Language code	Name
pal	Pahlavi
pam	Pampanga
pap	Papiamento
pau	Palauan
pcd	Picard
pcm	Nigerian Pidgin
pdc	Pennsylvania German
pdt	Plautdietsch
peo	Old Persian
pfl	Palatine German
phn	Phoenician
pi	Pali

Language code	Name
pl	Polish
pms	Piedmontese
pnt	Pontic
pon	Pohnpeian
pr	Pirate
prg	Prussian
pro	Old Provençal
prs	Dari
ps	Pushto
pt	Portuguese
qu	Quechua
quc	K'iche

Language code	Name
qug	Chimborazo Highland Quichua
quy	Ayacucho Quechua
quz	Cusco Quechua
raj	Rajasthani
rap	Rapanui
rar	Rarotongan
rgn	Romagnol
rif	Riffian
rm	Romansh
rn	Rundi
ro	Romanian
rof	Rombo

Language code	Name
rom	Romany
rtm	Rotuman
ru	Russian
rue	Rusyn
rug	Roviana
rup	Aromanian
rw	Kinyarwanda
rwk	Rwa
sa	Sanskrit
sad	Sandawe
sah	Sakha
sam	Samaritan Aramaic

Language code	Name
saq	Samburu
sas	Sasak
sat	Santali
saz	Saurashtra
sba	Ngambay
sbp	Sangu
sc	Sardinian
scn	Sicilian
sco	Scots
sd	Sindhi
sdc	Sassarese Sardinian

Language code	Name
sdh	Southern Kurdish
se	Northern Sami
see	Seneca
seh	Sena
sei	Seri
sel	Selkup
ses	Koyraboro Senni
sg	Sango
sga	Old Irish
sgs	Samogitian
sh	Serbo-Croatian
shi	Tachelhit

Language code	Name
shn	Shan
shs	Shuswap
shu	Chadian Arabic
si	Sinhala
sid	Sidamo
sk	Slovak
sl	Slovenian
sli	Lower Silesian
sly	Selayar
sm	Samoan
sma	Southern Sami
smj	Lule Sami

Language code	Name
smn	Inari Sami
sms	Skolt Sami
sn	Shona
snk	Soninke
so	Somali
sog	Sogdien
son	Songhai
sq	Albanian
sr	Serbian
srn	Sranan Tongo
srr	Serer
ss	Swati

Language code	Name
ssy	Saho
st	Southern Sotho
stq	Saterland Frisian
su	Sundanese
suk	Sukuma
sus	Susu
sux	Sumerian
sv	Swedish
sw	Swahili
swb	Comorian
swc	Congo Swahili
syc	Classical Syriac

Language code	Name
syr	Syriac
szl	Silesian
ta	Tamil
tcy	Tulu
te	Telugu
tem	Timne
teo	Teso
ter	Tereno
tet	Tetum
tg	Tajik
th	Thai

Language code	Name
the	Chitwania Tharu
ti	Tigrinya
tig	Tigre
tiv	Tiv
tk	Turkmen
tkl	Tokelau
tkr	Tsakhur
tl	Tagalog
tlh	Klingon
tli	Tlingit
tly	Talysh
tmh	Tamashek

Language code	Name
tn	Tswana
to	Tongan
tog	Nyasa Tonga
tpi	Tok Pisin
tr	Turkish
tru	Turoyo
trv	Taroko
ts	Tsonga
tsd	Tsakonian
tsi	Tsimshian
tt	Tatar
ttt	Muslim Tat

Language code	Name
tum	Tumbuka
tvl	Tuvalu
tw	Twi
twq	Tasawaq
ty	Tahitian
tyv	Tuvinian
tzm	Central Atlas Tamazight
udm	Udmurt
ug	Uyghur
uga	Ugaritic
uk	Ukrainian
umb	Umbundu

Language code	Name
unm	Unami
ur	Urdu
uz	Uzbek
vai	Vai
ve	Venda
vec	Venetian
vep	Veps
vi	Vietnamese
vls	West Flemish
vmf	Main-Franconian
vo	Volapük
vot	Votic

Language code	Name
vro	Võro
vun	Vunjo
wa	Walloon
wae	Walser
wal	Wolaytta
war	Waray
was	Washo
wbp	Warlpiri
wo	Wolof
wuu	Wu Chinese
xal	Kalmyk

Language code	Name
xh	Xhosa
xmf	Mingrelian
xog	Soga
yao	Yao
yap	Yapese
yav	Yangben
ybb	Yemba
yi	Yiddish
yo	Yoruba
yrl	Nheengatu
yue	Yue Chinese
yuw	Papua New Guinea

Language code	Name
za	Zhuang
zap	Zapotec
zbl	Blissymbols
zea	Zeelandic
zen	Zenaga
zgh	Standard Moroccan Tamazight
zh	Chinese
zu	Zulu
zun	Zuni
zza	Zaza

List of supported script codes

Script code	Name
Adlm	Adlam
Afak	Afaka
Aghb	Caucasian Albanian
Ahom	Ahom
Arab	Arabic
Armi	Imperial Aramaic
Armn	Armenian
Avst	Avestan
Bali	Balinese
Bamu	Bamum
Bass	Bassa Vah
Batk	Batak

Script code	Name
Beng	Bengali
Bhks	Bhaiksuki
Blis	Blissymbols
Bopo	Bopomofo
Brah	Brahmi
Brai	Braille
Bugi	Buginese
Buhd	Buhid
Cakm	Chakma
Cans	Unified Canadian Aboriginal
Cari	Carian
Cham	Cham

Script code	Name
Cher	Cherokee
Chrs	Chorasmian
Cirt	Cirth
Copt	Coptic
Cpmn	Cypro-Minoan
Cprt	Cypriot
Cyrl	Cyrillic
Deva	Devanagari
Diak	Dives Akuru
Dogr	Dogra
Dsrt	Deseret
Dupl	Duployan

Script code	Name
Egyd	Egyptian demotic
Egyh	Egyptian hieratic
Egyp	Egyptian hieroglyphs
Elba	Elbasan
Elym	Elymaic
Ethi	Ethiopic
Geok	Khutsuri
Geor	Georgian
Glag	Glagolitic
Gong	Gunjala Gondi
Gonm	Masaram Gondi
Goth	Gothic

Script code	Name
Gran	Grantha
Grek	Greek
Gujr	Gujarati
Guru	Gurmukhi
Hang	Hangul
Hani	Han
Hano	Hanunoo
Hans	Simplified
Hant	Traditional
Hatr	Hatran
Hebr	Hebrew

Script code	Name
Hira	Hiragana
Hluw	Anatolian Hieroglyphs
Hmng	Pahawh Hmong
Hmnp	Nyiakeng Puachue Hmong
Hung	Old Hungarian
Inds	Indus
Ital	Old Italic
Java	Javanese
Jurc	Jurchen
Kali	Kayah Li
Kana	Katakana
Khar	Kharoshthi

Script code	Name
Khmr	Khmer
Khoj	Khojki
Kitl	Khitan large script
Kits	Khitan small script
Knda	Kannada
Kpel	Kpelle
Kthi	Kaithi
Lana	Tai Tham
Laoo	Lao
Latn	Latin
Leke	Leke
Lepc	Lepcha

Script code	Name
Limb	Limbu
Lina	Linear A
Linb	Linear B
Lisu	Lisu
Loma	Loma
Lyci	Lycian
Lydi	Lydian
Mahj	Mahajani
Maka	Makasar
Mand	Mandaic
Mani	Manichaean
Marc	Marchen

Script code	Name
Maya	Mayan Hieroglyphs
Medf	Medefaidrin
Mend	Mende Kikakui
Merc	Meroitic Cursive
Mero	Meroitic Hieroglyphs
Mlym	Malayalam
Modi	Modi
Mong	Mongolian
Moon	Moon
Mroo	Mro
Mtei	Meitei Mayek
Mult	Multani

Script code	Name
Mymr	Myanmar (Burmese)
Nand	Nandinagari
Narb	Old North Arabian
Nbat	Nabataean
Newa	Newa
Nkdb	Naxi Dongba
Nkgb	Nakhi Geba
Nkoo	N'ko
Nshu	Nüshu
Ogam	Ogham
Olck	Ol Chiki

Script code	Name
Orkh	Old Turkic
Orya	Oriya
Osge	Osage
Osma	Osmanya
Ougr	Old Uyghur
Palm	Palmyrene
Pauc	Pau Cin Hau
Pcun	Proto-Cuneiform
Pelm	Proto-Elamite
Perm	Old Permic
Phag	Phags-pa
Phli	Inscriptional Pahlavi

Script code	Name
Phlp	Psalter Pahlavi
Phlv	Book Pahlavi
Phnx	Phoenician
Piqd	Klingon
Plrd	Miao
Prti	Inscriptional Parthian
Psin	Proto-Sinaitic
Ranj	Ranjana
Rjng	Rejang
Rohg	Hanifi Rohingya
Roro	Rongorongo
Runr	Runic

Script code	Name
Samr	Samaritan
Sara	Sarati
Sarb	Old South Arabian
Saur	Saurashtra
Sgnw	SignWriting
Shaw	Shavian
Shrd	Sharada
Shui	Shuishu
Sidd	Siddham
Sind	Khudawadi
Sinh	Sinhala
Sogd	Sogdian

Script code	Name
Sogo	Old Sogdian
Sora	Sora Sompeng
Soyo	Soyombo
Sund	Sundanese
Sylo	Syloti Nagri
Syrc	Syriac
Tagb	Tagbanwa
Takr	Takri
Tale	Tai Le
Talu	New Tai Lue
Taml	Tamil
Tang	Tangut

Script code	Name
Tavt	Tai Viet
Telu	Telugu
Teng	Tengwar
Tfng	Tifinagh
Tglg	Tagalog
Thaa	Thaana
Thai	Thai
Tibt	Tibetan
Tirh	Tirhuta
Tnsa	Tangsa
Toto	Toto

Script code	Name
Ugar	Ugaritic
Vaii	Vai
Visp	Visible Speech
Vith	Vithkuqi
Wara	Warang Citi
Wcho	Wancho
Wole	Woleai
Xpeo	Old Persian
Xsux	Cuneiform
Yezi	Yezidi
Yiii	Yi
Zanb	Zanabazar Square

Script code	Name

List of supported country codes

Country code	Name
AC	Ascension Island
AD	Andorra
AE	United Arab Emirates
AF	Afghanistan
AG	Antigua and Barbuda
AI	Anguilla
AL	Albania
AM	Armenia
AN	Netherlands Antilles

Country code	Name
AO	Angola
AQ	Antarctica
AR	Argentina
AS	American Samoa
AT	Austria
AU	Australia
AW	Aruba
AX	Åland Islands
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BD	Bangladesh

Country code	**Name**
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BH	Bahrain
BI	Burundi
BJ	Benin
BL	St. Barthélemy
BM	Bermuda
BN	Brunei
BO	Bolivia
BQ	Caribbean Netherlands
BR	Brazil

Country code	Name
BS	Bahamas
BT	Bhutan
BV	Bouvet Island
BW	Botswana
BY	Belarus
BZ	Belize
CA	Canada
CC	Cocos (Keeling) Islands
CD	Congo - Kinshasa
CF	Central African Republic
CG	Congo - Brazzaville
CH	Switzerland

Country code	Name
CI	Côte d'Ivoire
CK	Cook Islands
CL	Chile
CM	Cameroon
CN	China
CO	Colombia
CP	Clipperton Island
CR	Costa Rica
CQ	Island of Sark
CU	Cuba
CV	Cabo Verde
CW	Curaçao

Country code	Name
CX	Christmas Island
CY	Cyprus
CZ	Czechia
DE	Germany
DG	Diego Garcia
DJ	Djibouti
DK	Denmark
DM	Dominica
DO	Dominican Republic
DZ	Algeria
EA	Ceuta and Melilla

Country code	**Name**
EC	Ecuador
EE	Estonia
EG	Egypt
EH	Western Sahara
ER	Eritrea
ES	Spain
ET	Ethiopia
EU	European Union
EZ	Eurozone
FI	Finland
FJ	Fiji
FK	Falkland Islands

Country code	Name
FM	Micronesia
FO	Faroe Islands
FR	France
FX	France, Metropolitan
GA	Gabon
GB, UK	United Kingdom
GD	Grenada
GE	Georgia
GF	French Guiana
GG	Guernsey
GH	Ghana
GI	Gibraltar

Country code	Name
GL	Greenland
GM	Gambia
GN	Guinea
GP	Guadeloupe
GQ	Equatorial Guinea
GR	Greece
GS	South Georgia and South Sandwich Islands
GT	Guatemala
GU	Guam
GW	Guinea-Bissau
GY	Guyana
HK	Hong Kong

Country code	Name
HM	Heard Island and McDonald Islands
HN	Honduras
HR	Croatia
HT	Haiti
HU	Hungary
IC	Canary Islands
ID	Indonesia
IE	Ireland
IL	Israel
IM	Isle of Man
IN	India
IO	British Indian Ocean Territory

Country code	Name
IQ	Iraq
IR	Iran
IS	Iceland
IT	Italy
JE	Jersey
JM	Jamaica
JO	Jordan
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KH	Cambodia

Country code	Name
KI	Kiribati
KM	Comoros
KN	St. Kitts and Nevis
KP	North Korea
KR	South Korea
KW	Kuwait
KY	Cayman Islands
KZ	Kazakhstan
LA	Laos
LB	Lebanon
LC	St. Lucia
LI	Liechtenstein

Country code	Name
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
LY	Libya
MA	Morocco
MC	Monaco
MD	Moldova
ME	Montenegro
MF	St. Martin

Country code	**Name**
MG	Madagascar
MH	Marshall Islands
MK	North Macedonia
ML	Mali
MM, BU	Myanmar
MN	Mongolia
MO	Macao
MP	Northern Mariana Islands
MQ	Martinique
MR	Mauritania
MS	Montserrat
MT	Malta

Country code	**Name**
MU	Mauritius
MV	Maldives
MW	Malawi
MX	Mexico
MY	Malaysia
MZ	Mozambique
NA	Namibia
NC	New Caledonia
NE	Niger
NF	Norfolk Island
NG	Nigeria
NI	Nicaragua

Country code	**Name**
NL	Netherlands
NO	Norway
NP	Nepal
NR	Nauru
NU	Niue
NZ	New Zealand
OM	Oman
PA	Panama
PE	Peru
PF	French Polynesia
PG	Papua New Guinea

Country code	Name
PH	Philippines
PK	Pakistan
PL	Poland
PM	St. Pierre and Miquelon
PN	Pitcairn Islands
PR	Puerto Rico
PS	Palestine
PT	Portugal
PW	Palau
PY	Paraguay
QA	Qatar
RE	Réunion

Country code	Name
RO	Romania
RS	Serbia
RU	Russia
RW	Rwanda
SA	Saudi Arabia
SB	Solomon Islands
SC	Seychelles
SD	Sudan
SE	Sweden
SG	Singapore
SH	St. Helena, Ascension and Tristan da Cunha
SI	Slovenia

Country code	**Name**
SJ	Svalbard and Jan Mayen
SK	Slovakia
SL	Sierra Leone
SM	San Marino
SN	Senegal
SO	Somalia
SR	Suriname
SS	South Sudan
ST	Sao Tome and Principe
SV	El Salvador
SX	Sint Maarten
SY	Syria

Country code	Name
SZ	Eswatini
TA	Tristan da Cunha
TC	Turks and Caicos Islands
TD	Chad
TF	French Southern Territories
TG	Togo
TH	Thailand
TJ	Tajikistan
TK	Tokelau
TL, TP	Timor-Leste
TM	Turkmenistan
TN	Tunisia

Country code	Name
TO	Tonga
TR	Turkey
TT	Trinidad and Tobago
TV	Tuvalu
TW	Taiwan
TZ	Tanzania
UA	Ukraine
UG	Uganda
UM	U.S. Outlying Islands
US	United States of America
UY	Uruguay

Country code	Name
UZ	Uzbekistan
VA	Holy See
VC	St. Vincent and the Grenadines
VE	Venezuela
VG	British Virgin Islands
VI	U.S. Virgin Islands
VN	Viet Nam
VU	Vanuatu
WF	Wallis and Futuna
WS	Samoa
XK, KV	Kosovo
YE	Yemen

Country code	Name
YT	Mayotte
ZA	South Africa
ZM	Zambia
ZW	Zimbabwe

Pseudolocalization

Introduction

When creating a game, the process of localization usually starts when development has finished. This means that translations aren't available during development for testing whether the project is internationalized properly.

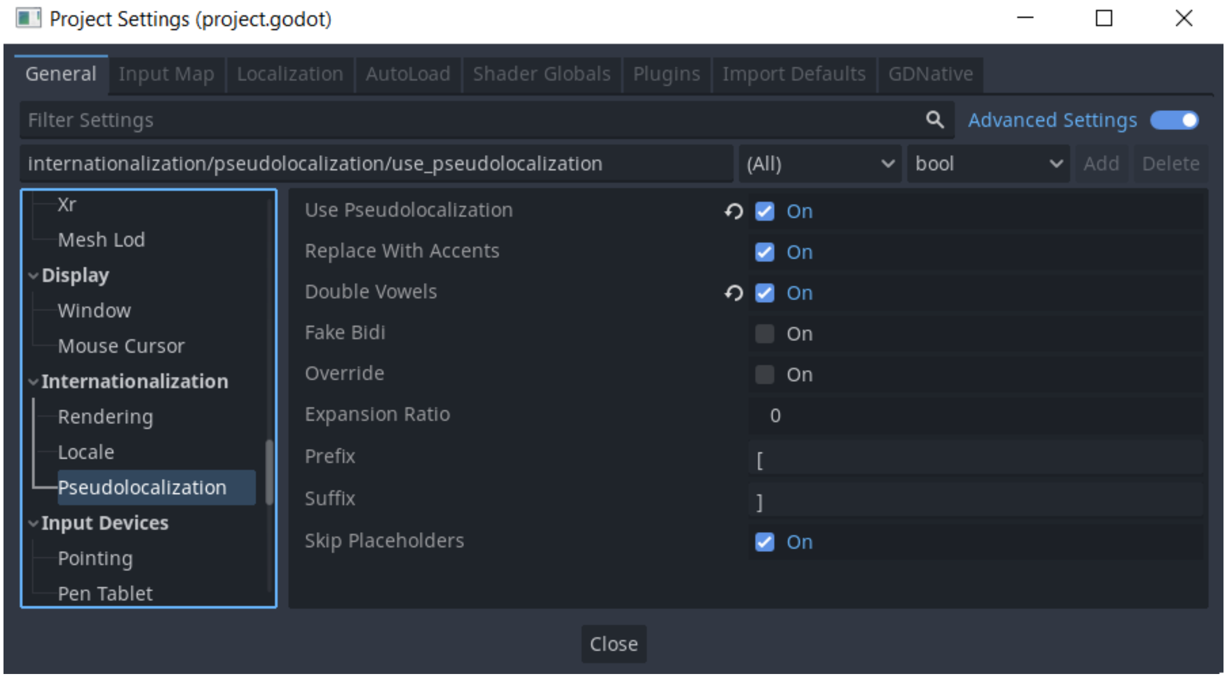
Godot offers pseudolocalization as a way to test how robust the project is when it comes to locale changes. Pseudolocalization simulates changes that might take place during localization. This way, any issues regarding internationalization can be recognized early on during development.

See also

You can see how pseudolocalization works in action using the Pseudolocalizaton demo project [https://github.com/godotengine/godot-demo-projects/tree/master/gui/pseudolocalization].

Enabling and configuring pseudolocalization

Enabling pseudolocalization and the configurations related to it is as simple as toggling a checkbox in the project settings. These settings can be found in **Project** → **Project Settings** → **General** → **Internationalization** → **Pseudolocalization** after enabling the **Advanced** toggle in the project settings dialog:


Project Settings (project.godot)
General
Input Map
Localization
AutoLoad
Shader Globals
Plugins
Import Defaults
GDNative
Filter Settings
Advanced Settings
(All)
bool
Add
Delete
Xr
Mesh Lod
Display
Window
Mouse Cursor
Internationalization
Rendering
Locale
Pseudolocalization
Input Devices
Pointing
Pen Tablet
Close

internationalization/pseudolocalization/use_pseudolocalization

Use Pseudolocalization ☑ On
Replace With Accents ☑ On
Double Vowels ☑ On
Fake Bidi ☐ On
Override ☐ On
Expansion Ratio 0
Prefix [
Suffix]
Skip Placeholders ☑ On

Pseudolocalization can also be toggled at runtime from a script.

Pseudolocalization configurations

Pseudolocalization in Godot can be set up according to the specific use case of the project. Here are the pseudolocalization properties that can be configured through project settings:

- `replace_with_accents`: Replaces all characters in the string with their accented variants. *"The quick brown fox jumped over the lazy dog"* will be converted to *"Ŧh̀é q́üíćḱ ḅŕôŵή f́ôx́ ǰüm̀ṕéd́ ôṽéŕ ŧh̀é łáźý d́ôǵ"* when this setting is enabled. This can be used to spot untranslated strings that won't have accents, but is also useful to check for missing glyphs in the font(s) used by the project.
- `double_vowels`: Doubles all the vowels in the string. It is a good approximation to simulate expansion of text during localization. This can be used to check for text that would overflow its container (such as buttons).

- `fake_bidi`: Fake bidirectional text (simulates right-to-left text). This is useful to simulate right-to-left writing systems to check for potential layout issues that would occur in languages using right-to-left scripts.
- `override`: Replaces all the characters in the string with an asterisk (`*`). This is useful for quickly finding text that isn't being localized.
- `expansion_ratio`: Can be used in cases where doubling the vowels isn't a sufficient approximation. This setting pads the string with underscores (_) and expands it by the given ratio. An expansion ratio of `0.3` is sufficient for most practical cases; it will increase the length of the string by 30%.
- `prefix` and `suffix`: These properties can be used to specify a prefix and suffix to wrap the text in.
- `skip_placeholders`: Skips placeholders for string formatting like `%s` and `%f`. This is useful to identify places where more arguments are required for the formatted string to display correctly.

All of these properties can be toggled as needed according to the project's use case.

Configuring pseudolocalization at runtime

Pseudolocalization can be toggled at runtime using the pseudolocalization_enabled property in TranslationServer. However, if runtime configuration of pseudolocalization properties is required, they can be directly configured using ProjectSettings.set_setting(property, value) and then calling TranslationServer.reload_pseudolocalization() which reparses the pseudolocalization properties and reloads the pseudolocalization. The following code snippet shall turn on `replace_with_accents` and `double_vowels` properties and then call `reload_pseudolocalization()` for the changes to get reflected:

```
ProjectSettings.set_setting("internationalization/pseudolocalization/replace_with_accents", true)
ProjectSettings.set_setting("internationalization/pseudolocalization/double_vowels", true)
TranslationServer.reload_pseudolocalization()
```

Input handling

- Using InputEvent
- Input examples
- Mouse and input coordinates
- Customizing the mouse cursor
- Controllers, gamepads, and joysticks
- Handling quit requests

Using InputEvent

What is it?

Managing input is usually complex, no matter the OS or platform. To ease this a little, a special built-in type is provided, InputEvent. This datatype can be configured to contain several types of input events. Input events travel through the engine and can be received in multiple locations, depending on the purpose.

Here is a quick example, closing your game if the escape key is hit:

GDScript

```
func _unhandled_input(event):
    if event is InputEventKey:
        if event.pressed and event.keycode == KEY_ESCAPE:
            get_tree().quit()
```

C#

```
public override void _UnhandledInput(InputEvent @event)
{
    if (@event is InputEventKey eventKey)
        if (eventKey.Pressed && eventKey.Keycode == Key.Escape)
            GetTree().Quit();
}
```

However, it is cleaner and more flexible to use the provided InputMap feature, which allows you to define input actions and assign them different keys. This way, you can define multiple keys for the same action (e.g. the keyboard escape key and the start button on a gamepad). You can then more easily change this mapping in the project settings without updating your code, and even build a key mapping feature on top of it to allow your game to change the key mapping at runtime!

You can set up your InputMap under **Project > Project Settings > Input Map** and then use those actions like this:

GDScript

```
func _process(delta):
    if Input.is_action_pressed("ui_right"):
        # Move right.
```

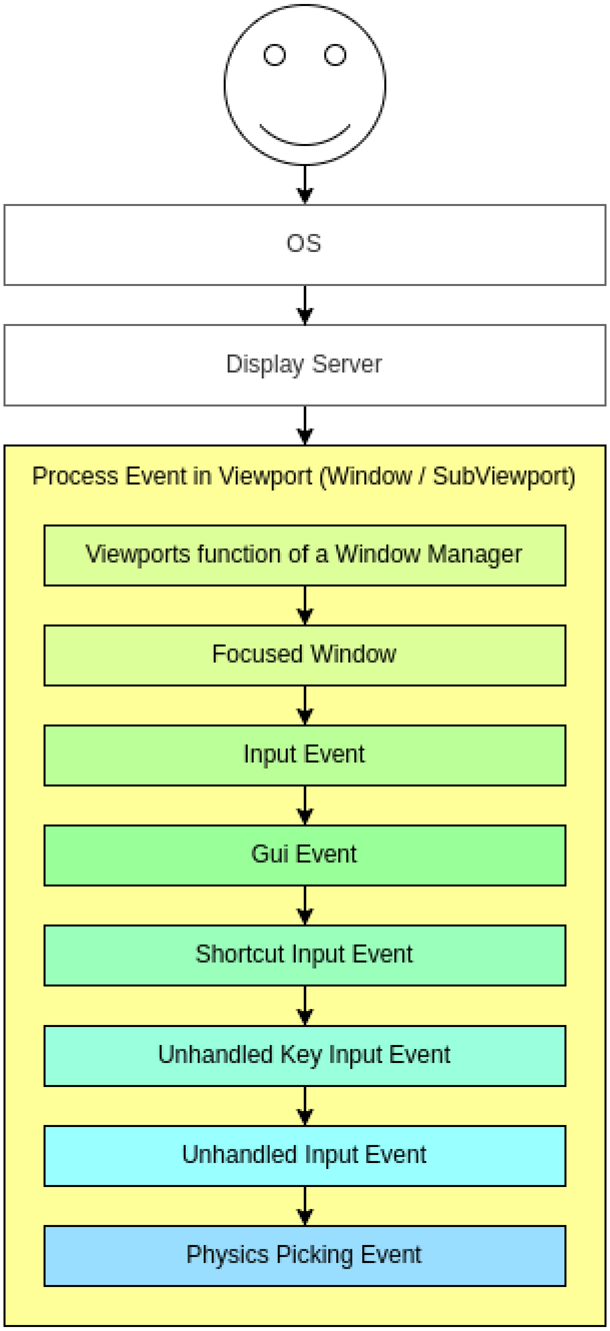
C#

```
public override void _Process(double delta)
{
    if (Input.IsActionPressed("ui_right"))
    {
        // Move right.
    }
}
```

How does it work?

Every input event is originated from the user/player (though it's possible to generate an InputEvent and feed them back to the engine, which is useful for gestures). The DisplayServer for each platform will read events from the operating system, then feed them to the root Window.

The window's Viewport does quite a lot of stuff with the received input, in order:


OS
Display Server
Process Event in Viewport (Window / SubViewport)
Viewports function of a Window Manager
Focused Window
Input Event
Gui Event
Shortcut Input Event
Unhandled Key Input Event
Unhandled Input Event
Physics Picking Event

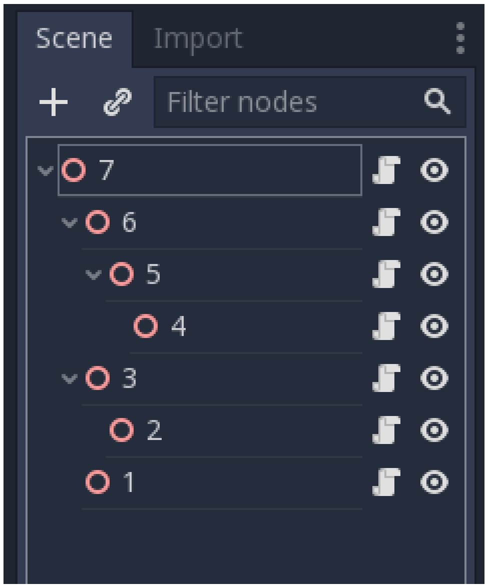
1. If the Viewport is embedding Windows, the Viewport tries to interpret the event in its capability as a Window-Manager (e.g. for resizing or moving Windows).
2. Next if an embedded Window is focused, the event is sent to that Window and processed in the Windows Viewport and afterwards treated as handled. If no embedded Window is

focused, the event is sent to the nodes of the current viewport in the following order.

3. First of all, the standard Node._input() function will be called in any node that overrides it (and hasn't disabled input processing with Node.set_process_input()). If any function consumes the event, it can call Viewport.set_input_as_handled(), and the event will not spread any more. This ensures that you can filter all events of interest, even before the GUI. For gameplay input, Node._unhandled_input() is generally a better fit, because it allows the GUI to intercept the events.
4. Second, it will try to feed the input to the GUI, and see if any control can receive it. If so, the Control will be called via the virtual function Control._gui_input() and the signal "gui_input" will be emitted (this function is re-implementable by script by inheriting from it). If the control wants to "consume" the event, it will call Control.accept_event() and the event will not spread any more. Use the Control.mouse_filter property to control whether a Control is notified of mouse events via Control._gui_input() callback, and whether these events are propagated further.
5. If so far no one consumed the event, the Node._shortcut_input() callback will be called if overridden (and not disabled with Node.set_process_shortcut_input()). This happens only for InputEventKey, InputEventShortcut and InputEventJoypadButton. If any function consumes the event, it can call Viewport.set_input_as_handled(), and the event will not spread any more. The shortcut input callback is ideal for treating events that are intended as shortcuts.
6. If so far no one consumed the event, the Node._unhandled_key_input() callback will be called if overridden (and not disabled with Node.set_process_unhandled_key_input()). This happens only if the event is an InputEventKey. If any function consumes the event, it can call Viewport.set_input_as_handled(), and the event will not spread any more. The unhandled key input callback is ideal for key events.

7. If so far no one consumed the event, the Node._unhandled_input() callback will be called if overridden (and not disabled with Node.set_process_unhandled_input()). If any function consumes the event, it can call Viewport.set_input_as_handled(), and the event will not spread any more. The unhandled input callback is ideal for full-screen gameplay events, so they are not received when a GUI is active.
8. If no one wanted the event so far, and Object Picking is turned on, the event is used for object picking. For the root viewport, this can also be enabled in Project Settings. In the case of a 3D scene if a Camera3D is assigned to the Viewport, a ray to the physics world (in the ray direction from the click) will be cast. If this ray hits an object, it will call the CollisionObject3D._input_event() function in the relevant physics object. In the case of a 2D scene, conceptually the same happens with CollisionObject2D._input_event().

When sending events to its child and descendant nodes, the viewport will do so, as depicted in the following graphic, in a reverse depth-first order, starting with the node at the bottom of the scene tree, and ending at the root node. Excluded from this process are Windows and SubViewports.



This order doesn't apply to Control._gui_input(), which uses a different method based on event location or focused Control.

Since Viewports don't send events to other SubViewports, one of the following methods has to be used:

1. Use a SubViewportContainer, which automatically sends events to its child SubViewports after Node._input() or Control._gui_input().
2. Implement event propagation based on the individual requirements.

GUI events also travel up the scene tree but, since these events target specific Controls, only direct ancestors of the targeted Control node receive the event.

In accordance with Godot's node-based design, this enables specialized child nodes to handle and consume particular events, while their ancestors, and ultimately the scene root, can provide more generalized behavior if needed.

Anatomy of an InputEvent

InputEvent is just a base built-in type, it does not represent anything and only contains some basic information, such as event ID (which is increased for each event), device index, etc.

There are several specialized types of InputEvent, described in the table below:

Event	Description
InputEvent	Empty Input Event.

InputEventKey	Contains a keycode and Unicode value, as well as modifiers.
InputEventMouseButton	Contains click information, such as button, modifiers, etc.
InputEventMouseMotion	Contains motion information, such as relative and absolute positions and speed.
InputEventJoypadMotion	Contains Joystick/Joypad analog axis information.
InputEventJoypadButton	Contains Joystick/Joypad button information.
InputEventScreenTouch	Contains multi-touch press/release information. (only available on mobile devices)
InputEventScreenDrag	Contains multi-touch drag information. (only

	available on mobile devices)
InputEventMagnifyGesture	Contains a position, a factor as well as modifiers.
InputEventPanGesture	Contains a position, a delta as well as modifiers.
InputEventMIDI	Contains MIDI-related information.
InputEventShortcut	Contains a shortcut.
InputEventAction	Contains a generic action. These events are often generated by the programmer as feedback. (more on this below)

Actions

Actions are a grouping of zero or more InputEvents into a commonly understood title (for example, the default "ui_left" action grouping both joypad-left input and a keyboard's left arrow key). They are not required to represent an InputEvent but are useful because they abstract various inputs when programming the game logic.

This allows for:

- The same code to work on different devices with different inputs (e.g., keyboard on PC, Joypad on console).
- Input to be reconfigured at run-time.
- Actions to be triggered programmatically at run-time.

Actions can be created from the Project Settings menu in the **Input Map** tab and assigned input events.

Any event has the methods InputEvent.is_action(), InputEvent.is_pressed() and InputEvent.

Alternatively, it may be desired to supply the game back with an action from the game code (a good example of this is detecting gestures). The Input singleton has a method for this: Input.parse_input_event(). You would normally use it like this:

GDScript

```
var ev = InputEventAction.new()
# Set as ui_left, pressed.
ev.action = "ui_left"
ev.pressed = true
# Feedback.
Input.parse_input_event(ev)
```

C#

```
var ev = new InputEventAction();
// Set as ui_left, pressed.
ev.Action = "ui_left";
ev.Pressed = true;
// Feedback.
Input.ParseInputEvent(ev);
```

InputMap

Customizing and re-mapping input from code is often desired. If your whole workflow depends on actions, the InputMap singleton is ideal for reassigning or creating different actions at run-time. This singleton is not saved (must be modified manually) and its state is run from the project settings (project.godot). So any dynamic system of this type needs to store settings in the way the programmer best sees fit.

Input examples

Introduction

In this tutorial, you'll learn how to use Godot's InputEvent system to capture player input. There are many different types of input your game may use - keyboard, gamepad, mouse, etc. - and many different ways to turn those inputs into actions in your game. This document will show you some of the most common scenarios, which you can use as starting points for your own projects.

Note

For a detailed overview of how Godot's input event system works, see Using InputEvent.

Events versus polling

Sometimes you want your game to respond to a certain input event - pressing the "jump" button, for example. For other situations, you might want something to happen as long as a key is pressed, such as movement. In the first case, you can use the `_input()` function, which will be called whenever an input event occurs. In the second case, Godot provides the Input singleton, which you can use to query the state of an input.

Examples:

GDScript

```
func _input(event):
    if event.is_action_pressed("jump"):
        jump()


func _physics_process(delta):
    if Input.is_action_pressed("move_right"):
        # Move as long as the key/button is pressed.
        position.x += speed * delta
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event.IsActionPressed("jump"))
    {
        Jump();
    }
}

public override void _PhysicsProcess(double delta)
{
    if (Input.IsActionPressed("move_right"))
    {
        // Move as long as the key/button is pressed.
        position.X += speed * (float)delta;
    }
}
```

This gives you the flexibility to mix-and-match the type of input processing you do.

For the remainder of this tutorial, we'll focus on capturing individual events in `_input()`.

Input events

Input events are objects that inherit from InputEvent. Depending on the event type, the object will contain specific properties related to that event. To see what events actually look like, add a Node and attach the following script:

GDScript

```
extends Node


func _input(event):
    print(event.as_text())
```

C#

```
using Godot;

public partial class Node : Godot.Node
{
    public override void _Input(InputEvent @event)
    {
        GD.Print(@event.AsText());
    }
}
```

As you press keys, move the mouse, and perform other inputs, you'll see each event scroll by in the output window. Here's an example of the output:

```
A
Mouse motion at position ((971, 5)) with velocity ((0, 0))
Right Mouse Button
Mouse motion at position ((870, 243)) with velocity ((0.454937,
-0.454937))
Left Mouse Button
Mouse Wheel Up
A
B
Shift
Alt+Shift
Alt
Shift+T
```

```
Mouse motion at position ((868, 242)) with velocity
((-2.134768, 2.134768))
```

As you can see, the results are very different for the different types of input. Key events are even printed as their key symbols. For example, let's consider InputEventMouseButton. It inherits from the following classes:

- InputEvent - the base class for all input events
- InputEventWithModifiers - adds the ability to check if modifiers are pressed, such as `Shift` or `Alt`.
- InputEventMouse - adds mouse event properties, such as `position`
- InputEventMouseButton - contains the index of the button that was pressed, whether it was a double-click, etc.

Tip

It's a good idea to keep the class reference open while you're working with events so you can check the event type's available properties and methods.

You can encounter errors if you try to access a property on an input type that doesn't contain it - calling `position` on `InputEventKey` for example. To avoid this, make sure to test the event type first:

GDScript

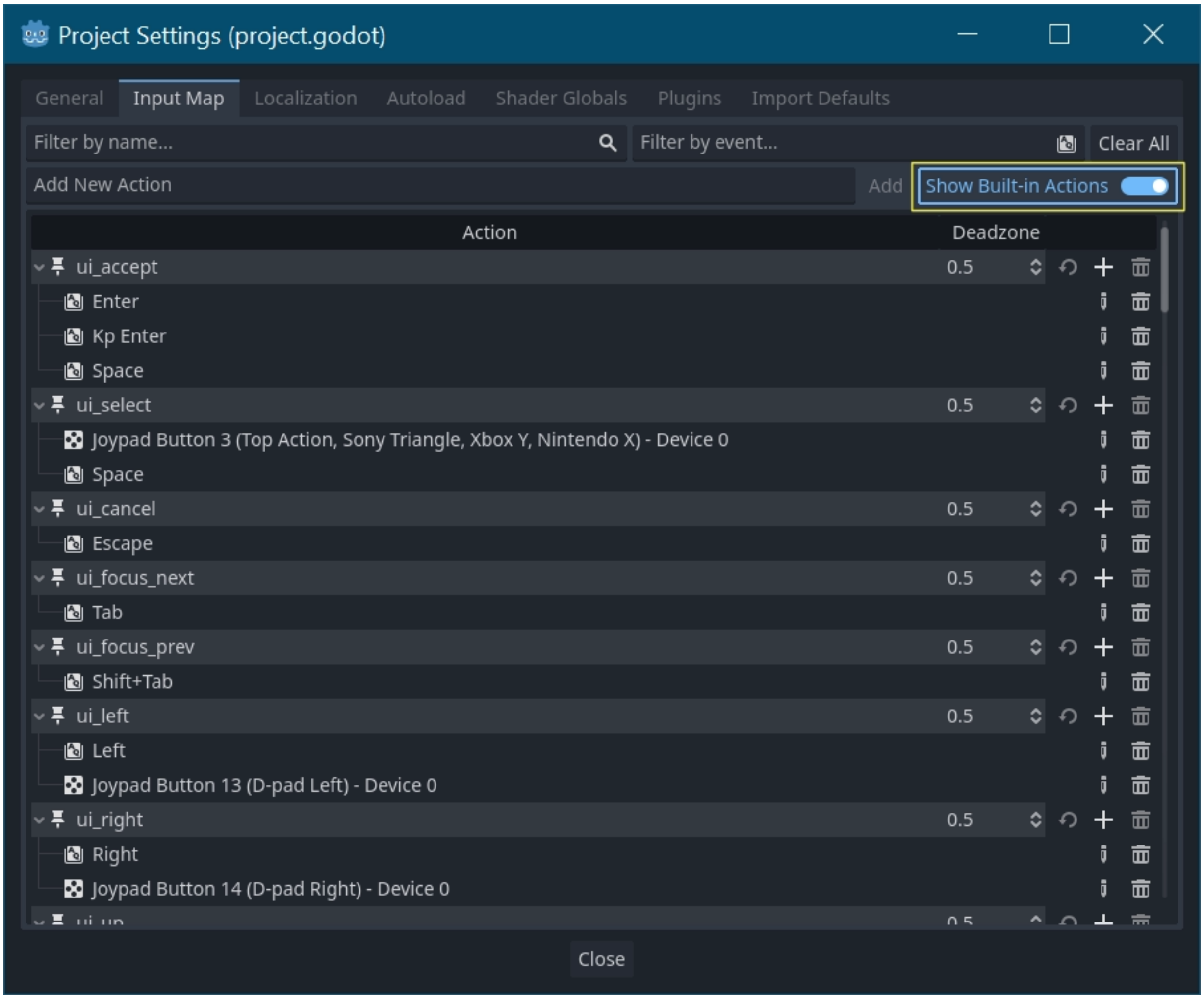
```
func _input(event):
    if event is InputEventMouseButton:
        print("mouse button event at ", event.position)
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event is InputEventMouseButton mouseEvent)
    {
        GD.Print("mouse button event at ", 
mouseEvent.Position);
    }
}
```

InputMap

The InputMap is the most flexible way to handle a variety of inputs. You use this by creating named input *actions*, to which you can assign any number of input events, such as keypresses or mouse clicks. To see them, and to add your own, open Project -> Project Settings and select the InputMap tab:


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Filter by name...
Filter by event...
Clear All
Add New Action
Add
Show Built-in Actions
Close

Action	Deadzone
ui_accept	0.5
Enter	
Kp Enter	
Space	
ui_select	0.5
Joypad Button 3 (Top Action, Sony Triangle, Xbox Y, Nintendo X) - Device 0	
Space	
ui_cancel	0.5
Escape	
ui_focus_next	0.5
Tab	
ui_focus_prev	0.5
Shift+Tab	
ui_left	0.5
Left	
Joypad Button 13 (D-pad Left) - Device 0	
ui_right	0.5
Right	
Joypad Button 14 (D-pad Right) - Device 0	

Tip

A new Godot project includes a number of default actions already defined. To see them, turn on `Show Built-in Actions` in the InputMap dialog.

Capturing actions

Once you've defined your actions, you can process them in your scripts using `is_action_pressed()` and `is_action_released()` by passing the name of the action you're looking for:

GDScript

```
func _input(event):
    if event.is_action_pressed("my_action"):
        print("my_action occurred!")
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event.IsActionPressed("my_action"))
    {
        GD.Print("my_action occurred!");
    }
}
```

Keyboard events

Keyboard events are captured in InputEventKey. While it's recommended to use input actions instead, there may be cases where you want to specifically look at key events. For this example, let's check for the `T`:

GDScript

```
func _input(event):
    if event is InputEventKey and event.pressed:
        if event.keycode == KEY_T:
            print("T was pressed")
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event is InputEventKey keyEvent && keyEvent.Pressed)
    {
        if (keyEvent.Keycode == Key.T)
        {
            GD.Print("T was pressed");
        }
    }
}
```

Tip

See @GlobalScope_Key for a list of keycode constants.

Warning

Due to *keyboard ghosting*, not all key inputs may be registered at a given time if you press too many keys at once. Due to their location on the keyboard, certain keys are more prone to ghosting than others. Some keyboards feature antighosting at a hardware level, but this feature is generally not present on low-end keyboards and laptop keyboards.

As a result, it's recommended to use a default keyboard layout that is designed to work well on a keyboard without antighosting. See this Gamedev Stack Exchange question [https://gamedev.stackexchange.com/a/109002] for more information.

Keyboard modifiers

Modifier properties are inherited from InputEventWithModifiers. This allows you to check for modifier combinations using boolean properties. Let's imagine you want one thing to happen when the `T` is pressed, but something different when it's `Shift + T`:

GDScript

```
func _input(event):
    if event is InputEventKey and event.pressed:
        if event.keycode == KEY_T:
            if event.shift_pressed:
                print("Shift+T was pressed")
            else:
                print("T was pressed")
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event is InputEventKey keyEvent && keyEvent.Pressed)
    {
        switch (keyEvent.Keycode)
        {
            case Key.T:
                GD.Print(keyEvent.ShiftPressed ? "Shift+T was pressed" : "T was pressed");
                break;
        }
    }
}
```

Tip

See @GlobalScope_Key for a list of keycode constants.

Mouse events

Mouse events stem from the InputEventMouse class, and are separated into two types: InputEventMouseButton and InputEventMouseMotion. Note that this means that all mouse events will contain a `position` property.

Mouse buttons

Capturing mouse buttons is very similar to handling key events. @GlobalScope_MouseButton contains a list of `MOUSE_BUTTON_*` constants for each possible button, which will be reported in the event's `button_index` property. Note that the scrollwheel also counts as a button - two buttons, to be precise, with both `MOUSE_BUTTON_WHEEL_UP` and `MOUSE_BUTTON_WHEEL_DOWN` being separate events.

GDScript

```
func _input(event):
    if event is InputEventMouseButton:
        if event.button_index == MOUSE_BUTTON_LEFT and
event.pressed:
            print("Left button was clicked at ",
event.position)
        if event.button_index == MOUSE_BUTTON_WHEEL_UP and
event.pressed:
            print("Wheel up")
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event is InputEventMouseButton mouseEvent &&
mouseEvent.Pressed)
    {
        switch (mouseEvent.ButtonIndex)
        {
            case MouseButton.Left:
                GD.Print($"Left button was clicked at
{mouseEvent.Position}");
                break;
            case MouseButton.WheelUp:
                GD.Print("Wheel up");
                break;
        }
    }
}
```

Mouse motion

InputEventMouseMotion events occur whenever the mouse moves. You can find the move's distance with the `relative` property.

Here's an example using mouse events to drag-and-drop a Sprite2D node:

GDScript

```
extends Node


var dragging = false
var click_radius = 32 # Size of the sprite.


func _input(event):
    if event is InputEventMouseButton and event.button_index ==
MOUSE_BUTTON_LEFT:
        if (event.position - $Sprite2D.position).length() <
click_radius:
            # Start dragging if the click is on the sprite.
            if not dragging and event.pressed:
                dragging = true
        # Stop dragging if the button is released.
        if dragging and not event.pressed:
            dragging = false

    if event is InputEventMouseMotion and dragging:
        # While dragging, move the sprite with the mouse.
        $Sprite2D.position = event.position
```

C#

```
using Godot;

public partial class MyNode2D : Node2D
{
    private bool _dragging = false;
    private int _clickRadius = 32; // Size of the sprite.
    public override void _Input(InputEvent @event)
    {
        Sprite2D sprite = GetNodeOrNull<Sprite2D>("Sprite2D");
        if (sprite == null)
        {
            return; // No suitable node was found.
        }

        if (@event is InputEventMouseButton mouseEvent &&
mouseEvent.ButtonIndex == MouseButton.Left)
        {
            if ((mouseEvent.Position -
sprite.Position).Length() < _clickRadius)
            {
                // Start dragging if the click is on the
sprite.
                if (!_dragging && mouseEvent.Pressed)
                {
                    _dragging = true;
                }
            }
            // Stop dragging if the button is released.
            if (_dragging && !mouseEvent.Pressed)
            {
                _dragging = false;
            }
        }
        else
        {
            if (@event is InputEventMouseMotion motionEvent &&
_dragging)
            {
                // While dragging, move the sprite with the
mouse.
                sprite.Position = motionEvent.Position;
            }
        }
    }
}
```

Touch events

If you are using a touchscreen device, you can generate touch events. InputEventScreenTouch is equivalent to a mouse click event, and InputEventScreenDrag works much the same as mouse motion.

Tip

To test your touch events on a non-touchscreen device, open Project Settings and go to the "Input Devices/Pointing" section. Enable "Emulate Touch From Mouse" and your project will interpret mouse clicks and motion as touch events.

Mouse and input coordinates

About

The reason for this small tutorial is to clear up many common mistakes about input coordinates, obtaining mouse position and screen resolution, etc.

Hardware display coordinates

Using hardware coordinates makes sense in the case of writing complex UIs meant to run on PC, such as editors, MMOs, tools, etc. However, it does not make as much sense outside of that scope.

Viewport display coordinates

Godot uses viewports to display content, and viewports can be scaled by several options (see Multiple resolutions tutorial). Use, then, the functions in nodes to obtain the mouse coordinates and viewport size, for example:

GDScript

```
func _input(event):
    # Mouse in viewport coordinates.
    if event is InputEventMouseButton:
        print("Mouse Click/Unclick at: ", event.position)
    elif event is InputEventMouseMotion:
        print("Mouse Motion at: ", event.position)

    # Print the size of the viewport.
    print("Viewport Resolution is: ",
get_viewport().get_visible_rect().size)
```

C#

```
public override void _Input(InputEvent @event)
{
    // Mouse in viewport coordinates.
    if (@event is InputEventMouseButton eventMouseButton)
        GD.Print("Mouse Click/Unclick at: ",
eventMouseButton.Position);
    else if (@event is InputEventMouseMotion eventMouseMotion)
        GD.Print("Mouse Motion at: ",
eventMouseMotion.Position);

    // Print the size of the viewport.
    GD.Print("Viewport Resolution is: ",
GetViewport().GetVisibleRect().Size);
}
```

Alternatively, it's possible to ask the viewport for the mouse position:

GDScript

```
get_viewport().get_mouse_position()
```

C#

```
GetViewport().GetMousePosition();
```

Note

When the mouse mode is set to `Input.MOUSE_MODE_CAPTURED`, the `event.position` value from `InputEventMouseMotion` is the center of the screen. Use `event.relative` instead of `event.position` and `event.velocity` to process mouse movement and position changes.

Customizing the mouse cursor

You might want to change the appearance of the mouse cursor in your game in order to suit the overall design. There are two ways to customize the mouse cursor:

1. Using project settings
2. Using a script

Using project settings is a simpler (but more limited) way to customize the mouse cursor. The second way is more customizable, but involves scripting:

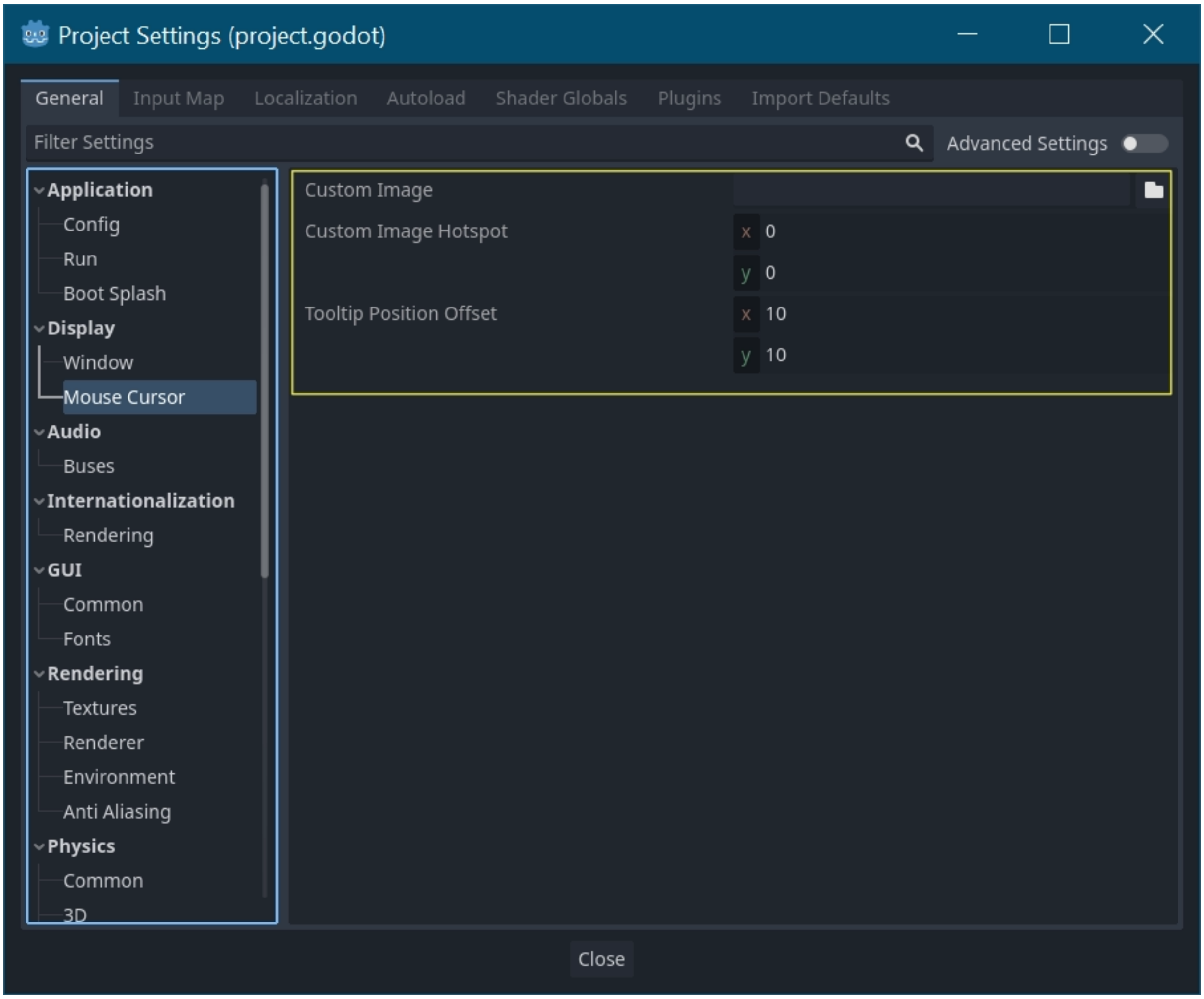
Note

You could display a "software" mouse cursor by hiding the mouse cursor and moving a Sprite2D to the cursor position in a `_process()` method, but this will add at least one frame of latency compared to an "hardware" mouse cursor. Therefore, it's recommended to use the approach described here whenever possible.

If you have to use the "software" approach, consider adding an extrapolation step to better display the actual mouse input.

Using project settings

Open project settings, go to Display>Mouse Cursor. You will see Custom Image, Custom Image Hotspot and Tooltip Position Offset.


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Filter Settings
Advanced Settings
Application
Config
Run
Boot Splash
Display
Window
Mouse Cursor
Audio
Buses
Internationalization
Rendering
GUI
Common
Fonts
Rendering
Textures
Renderer
Environment
Anti Aliasing
Physics
Common
3D
Close

Custom Image

Custom Image Hotspot x 0 y 0

Tooltip Position Offset x 10 y 10

Custom Image is the desired image that you would like to set as the mouse cursor. Custom Hotspot is the point in the image that you would like to use as the cursor's detection point.

Warning

The custom image **must** be 256×256 pixels at most. To avoid rendering issues, sizes lower than or equal to 128×128 are recommended.

On the web platform, the maximum allowed cursor image size is 128×128.

Using a script

Create a Node and attach the following script.

GDScript

```
extends Node


# Load the custom images for the mouse cursor.
var arrow = load("res://arrow.png")
var beam = load("res://beam.png")


func _ready():
    # Changes only the arrow shape of the cursor.
    # This is similar to changing it in the project settings.
    Input.set_custom_mouse_cursor(arrow)

    # Changes a specific shape of the cursor (here, the I-beam
shape).
    Input.set_custom_mouse_cursor(beam, Input.CURSOR_IBEAM)
```

C#

```
public override void _Ready()
{
    // Load the custom images for the mouse cursor.
    var arrow = ResourceLoader.Load("res://arrow.png");
    var beam = ResourceLoader.Load("res://beam.png");

    // Changes only the arrow shape of the cursor.
    // This is similar to changing it in the project settings.
    Input.SetCustomMouseCursor(arrow);

    // Changes a specific shape of the cursor (here, the I-beam
shape).
    Input.SetCustomMouseCursor(beam, Input.CursorShape.Ibeam);
}
```

See also

Check Input.set_custom_mouse_cursor()'s documentation for more information on usage and platform-specific caveats.

Cursor list

As documented in the Input class (see the **CursorShape** enum), there are multiple mouse cursors you can define. Which ones you want to use depends on your use case.

Controllers, gamepads, and joysticks

Godot supports hundreds of controller models thanks to the community-sourced SDL game controller database [https://github.com/gabomdq/SDL_GameControllerDB].

Controllers are supported on Windows, macOS, Linux, Android, iOS, and HTML5.

Note that more specialized devices such as steering wheels, rudder pedals and HOTAS [https://en.wikipedia.org/wiki/HOTAS] are less tested and may not always work as expected. Overriding force feedback for those devices is also not implemented yet. If you have access to one of those devices, don't hesitate to report bugs on GitHub [https://github.com/godotengine/godot/blob/master/CONTRIBUTING.md#reporting-bugs].

In this guide, you will learn:

- **How to write your input logic to support both keyboard and controller inputs.**
- **How controllers can behave differently from keyboard/mouse input.**
- **Troubleshooting issues with controllers in Godot.**

Supporting universal input

Thanks to Godot's input action system, Godot makes it possible to support both keyboard and controller input without having to write separate code paths. Instead of hardcoding keys or controller buttons in your scripts, you should create *input actions* in the Project Settings which will then refer to specified key and controller inputs.

Input actions are explained in detail on the Using InputEvent page.

Note

Unlike keyboard input, supporting both mouse and controller input for an action (such as looking around in a first-person game) will require different code paths since these have to be handled separately.

Which Input singleton method should I use?

There are 3 ways to get input in an analog-aware way:

- When you have two axes (such as joystick or WASD movement) and want both axes to behave as a single input, use `Input.get_vector()`:

GDScript

```
# `velocity` will be a Vector2 between `Vector2(-1.0, -1.0)`
and `Vector2(1.0, 1.0)`.
# This handles deadzone in a correct way for most use cases.
# The resulting deadzone will have a circular shape as it
generally should.
var velocity = Input.get_vector("move_left", "move_right",
"move_forward", "move_back")

# The line below is similar to `get_vector()`, except that it
handles
# the deadzone in a less optimal way. The resulting deadzone
will have
# a square-ish shape when it should ideally have a circular
shape.
var velocity = Vector2(
        Input.get_action_strength("move_right") -
Input.get_action_strength("move_left"),
        Input.get_action_strength("move_back") -
Input.get_action_strength("move_forward")
).limit_length(1.0)
```

C#

```
// `velocity` will be a Vector2 between `Vector2(-1.0, -1.0)`
and `Vector2(1.0, 1.0)`.
// This handles deadzone in a correct way for most use cases.
// The resulting deadzone will have a circular shape as it
generally should.
Vector2 velocity = Input.GetVector("move_left", "move_right",
"move_forward", "move_back");

// The line below is similar to `get_vector()`, except that it
handles
// the deadzone in a less optimal way. The resulting deadzone
will have
// a square-ish shape when it should ideally have a circular
shape.
Vector2 velocity = new Vector2(
        Input.GetActionStrength("move_right") -
Input.GetActionStrength("move_left"),
        Input.GetActionStrength("move_back") -
Input.GetActionStrength("move_forward")
).LimitLength(1.0);
```

- When you have one axis that can go both ways (such as a throttle on a flight stick), or when you want to handle separate axes individually, use `Input.get_axis()`:

GDScript

```
# `walk` will be a floating-point number between `-1.0` and
`1.0`.
var walk = Input.get_axis("move_left", "move_right")

# The line above is a shorter form of:
```

```
var walk = Input.get_action_strength("move_right") -
Input.get_action_strength("move_left")
```

C#

```
// `walk` will be a floating-point number between `-1.0` and
`1.0`.
float walk = Input.GetAxis("move_left", "move_right");

// The line above is a shorter form of:
float walk = Input.GetActionStrength("move_right") -
Input.GetActionStrength("move_left");
```

- For other types of analog input, such as handling a trigger or handling one direction at a time, use `Input.get_action_strength()`:

GDScript

```
# `strength` will be a floating-point number between `0.0` and
`1.0`.
var strength = Input.get_action_strength("accelerate")
```

C#

```
// `strength` will be a floating-point number between `0.0` and
`1.0`.
float strength = Input.GetActionStrength("accelerate");
```

For non-analog digital/boolean input (only "pressed" or "not pressed" values), such as controller buttons, mouse buttons or keyboard keys, use `Input.is_action_pressed()`:

GDScript

```
# `jumping` will be a boolean with a value of `true` or
`false`.
var jumping = Input.is_action_pressed("jump")
```

C#

```
// `jumping` will be a boolean with a value of `true` or
`false`.
bool jumping = Input.IsActionPressed("jump");
```

Note

If you need to know whether an input was *just* pressed in the previous frame, use `Input.is_action_just_pressed()` instead of `Input.is_action_pressed()`. Unlike `Input.is_action_pressed()` which returns `true` as long as the input is held, `Input.is_action_just_pressed()` will only return `true` for one frame after the button has been pressed.

In Godot versions before 3.4, such as 3.3, `Input.get_vector()` and `Input.get_axis()` aren't available. Only `Input.get_action_strength()` and `Input.is_action_pressed()` are available in Godot 3.3.

Vibration

Vibration (also called *haptic feedback*) can be used to enhance the feel of a game. For instance, in a racing game, you can convey the surface the car is currently driving on through vibration, or create a sudden vibration on a crash.

Use the Input singleton's start_joy_vibration method to start vibrating a gamepad. Use stop_joy_vibration to stop vibration early (useful if no duration was specified when starting).

On mobile devices, you can also use vibrate_handheld to vibrate the device itself (independently from the gamepad). On Android, this requires the `VIBRATE` permission to be enabled in the Android export preset before exporting the project.

Note

Vibration can be uncomfortable for certain players. Make sure to provide an in-game slider to disable vibration or reduce its intensity.

Differences between keyboard/mouse and controller input

If you're used to handling keyboard and mouse input, you may be surprised by how controllers handle specific situations.

Dead zone

Unlike keyboards and mice, controllers offer axes with *analog* inputs. The upside of analog inputs is that they offer additional flexibility for actions. Unlike digital inputs which can only provide strengths of `0.0` and `1.0`, an analog input can provide *any* strength between `0.0` and `1.0`. The downside is that without a deadzone system, an analog axis' strength will never be equal to `0.0` due to how the controller is physically built. Instead, it will linger at a low value such as `0.062`. This phenomenon is known as *drifting* and can be more noticeable on old or faulty controllers.

Let's take a racing game as a real-world example. Thanks to analog inputs, we can steer the car slowly in one direction or another. However, without a deadzone system, the car would slowly steer by itself even if the player isn't touching the joystick. This is because the directional axis strength won't be equal to `0.0` when we expect it to. Since we don't want our car to steer by itself in this case, we define a "dead zone" value of `0.2` which will ignore all input whose strength is lower than `0.2`. An ideal dead zone value is high enough to ignore the input caused by joystick drifting, but is low enough to not ignore actual input from the player.

Godot features a built-in deadzone system to tackle this problem. The default value is `0.5`, but you can adjust it on a per-action basis in the Project Settings' Input Map tab. For `Input.get_vector()`, the deadzone can be specified as an optional 5th parameter. If not specified, it will calculate the average deadzone value from all of the actions in the vector.

"Echo" events

Unlike keyboard input, holding down a controller button such as a D-pad direction will **not** generate repeated input events at fixed intervals (also known as "echo" events). This is because the operating system never sends "echo" events for controller input in the first place.

If you want controller buttons to send echo events, you will have to generate InputEvent objects by code and parse them using Input.parse_input_event() at regular intervals. This can be accomplished with the help of a Timer node.

Window focus

Unlike keyboard input, controller inputs can be seen by **all** windows on the operating system, including unfocused windows.

While this is useful for third-party split screen functionality [https://nucleus-coop.github.io/], it can also have adverse effects. Players may accidentally send controller inputs to the running project while interacting with another window.

If you wish to ignore events when the project window isn't focused, you will need to create an autoload called `Focus` with the following script and use it to check all your inputs:

```
# Focus.gd
extends Node

var focused := true

func _notification(what: int) -> void:
    match what:
        NOTIFICATION_APPLICATION_FOCUS_OUT:
            focused = false
        NOTIFICATION_APPLICATION_FOCUS_IN:
            focused = true


func input_is_action_pressed(action: StringName) -> bool:
    if focused:
        return Input.is_action_pressed(action)

    return false


func event_is_action_pressed(event: InputEvent, action:
StringName) -> bool:
    if focused:
        return event.is_action_pressed(action)

    return false
```

Then, instead of using `Input.is_action_pressed(action)`, use `Focus.input_is_action_pressed(action)` where `action` is the name of the input action. Also, instead of using `event.is_action_pressed(action)`, use `Focus.event_is_action_pressed(event, action)` where `event` is an InputEvent reference and `action` is the name of the input action.

Power saving prevention

Unlike keyboard and mouse input, controller inputs do **not** inhibit sleep and power saving measures (such as turning off the screen after a certain amount of time has passed).

To combat this, Godot enables power saving prevention by default when a project is running. If you notice the system is turning off its display when playing with a gamepad, check the value of **Display > Window > Energy Saving > Keep Screen On** in the Project Settings.

On Linux, power saving prevention requires the engine to be able to use D-Bus. Check whether D-Bus is installed and reachable if running the project within a Flatpak, as sandboxing restrictions may make this impossible by default.

Troubleshooting

See also

You can view a list of known issues with controller support [https://github.com/godotengine/godot/issues?q=is%3Aopen+is%3Aissue+label%3Atopic%3Ainput+gamepad] on GitHub.

My controller isn't recognized by Godot.

First, check that your controller is recognized by other applications. You can use the Gamepad Tester [https://gamepad-tester.com/] website to confirm that your controller is recognized.

On Windows Godot only supports up to 4 controllers at a time. This is because Godot uses the XInput API, which is limited to supporting 4 controllers at once. Additional controllers above this limit are ignored by Godot.

My controller has incorrectly mapped buttons or axes.

First, if your controller provides some kind of firmware update utility, make sure to run it to get the latest fixes from the manufacturer. For instance, Xbox One and Xbox Series controllers can have their firmware updated using the Xbox Accessories app [https://www.microsoft.com/en-us/p/xbox-accessories/9nblggh30xj3]. (This application only runs on Windows, so you have to use a Windows machine or a Windows virtual machine with USB support to update the controller's firmware.) After updating the controller's firmware, unpair the controller and pair it again with your PC if you are using the controller in wireless mode.

If buttons are incorrectly mapped, this may be due to an erroneous mapping from the SDL game controller database [https://github.com/gabomdq/SDL_GameControllerDB]. You can contribute an updated mapping to be included in the next Godot version by opening a pull request on the linked repository.

There are many ways to create mappings. One option is to use the mapping wizard in the official Joypads demo [https://godotengine.org/asset-library/asset/2785]. Once you have a working mapping for your controller, you can test it by defining the `SDL_GAMECONTROLLERCONFIG` environment variable before running Godot:

Linux/macOS

```
export SDL_GAMECONTROLLERCONFIG="your:mapping:here"
./path/to/godot.x86_64
```

Windows (cmd)

```
set SDL_GAMECONTROLLERCONFIG=your:mapping:here
path\to\godot.exe
```

Windows (PowerShell)

```
$env:SDL_GAMECONTROLLERCONFIG="your:mapping:here"
path\to\godot.exe
```

To test mappings on non-desktop platforms or to distribute your project with additional controller mappings, you can add them by calling Input.add_joy_mapping() as early as possible in a script's `_ready()` function.

My controller works on a given platform, but not on another platform.

Linux

If you're using a self-compiled engine binary, make sure it was compiled with udev support. This is enabled by default, but it is possible to disable udev support by specifying `udev=no` on the SCons command line. If you're using an engine binary supplied by a Linux distribution, double-check whether it was compiled with udev support.

Controllers can still work without udev support, but it is less reliable as regular polling must be used to check for controllers being connected or disconnected during gameplay (hotplugging).

HTML5

HTML5 controller support is often less reliable compared to "native" platforms. The quality of controller support tends to vary wildly across browsers. As a result, you may have to instruct your players to use a different browser if they can't get their controller to work.

Handling quit requests

Quitting

Most platforms have the option to request the application to quit. On desktops, this is usually done with the "x" icon on the window title bar. On mobile devices, the app can quit at any time while it is suspended to the background.

Handling the notification

On desktop and web platforms, Node receives a special `NOTIFICATION_WM_CLOSE_REQUEST` notification when quitting is requested from the window manager.

Handling the notification is done as follows (on any node):

GDScript

```
func _notification(what):
    if what == NOTIFICATION_WM_CLOSE_REQUEST:
        get_tree().quit() # default behavior
```

C#

```
public override void _Notification(int what)
{
    if (what == NotificationWMCloseRequest)
        GetTree().Quit(); // default behavior
}
```

It is important to note that by default, Godot apps have the built-in behavior to quit when quit is requested from the window manager.

This can be changed, so that the user can take care of the complete quitting procedure:

GDScript

```
get_tree().set_auto_accept_quit(false)
```

C#

```
GetTree().AutoAcceptQuit = false;
```

On mobile devices

There is no direct equivalent to `NOTIFICATION_WM_CLOSE_REQUEST` on mobile platforms. Due to the nature of mobile operating systems, the only place that you can run code prior to quitting is when the app is being suspended to the background. On both Android and iOS, the app can be killed while suspended at any time by either the user or the OS. A way to plan ahead for this possibility is to utilize `NOTIFICATION_APPLICATION_PAUSED` in order to perform any needed actions as the app is being suspended.

On Android, pressing the Back button will exit the application if **Application > Config > Quit** On Go Back is checked in the Project Settings (which is the default). This will fire `NOTIFICATION_WM_GO_BACK_REQUEST`.

Sending your own quit notification

While forcing the application to close can be done by calling SceneTree.quit, doing so will not send the `NOTIFICATION_WM_CLOSE_REQUEST` to the nodes in the scene tree. Quitting by calling SceneTree.quit will not allow custom actions to complete (such as saving, confirming the quit, or debugging), even if you try to delay the line that forces the quit.

Instead, if you want to notify the nodes in the scene tree about the upcoming program termination, you should send the notification yourself:

GDScript

```
get_tree().root.propagate_notification(NOTIFICATION_WM_CLOSE_RE
QUEST)
```

C#

```
GetTree().Root.PropagateNotification((int)NotificationWMCloseRe
quest);
```

Sending this notification will inform all nodes about the program termination, but will not terminate the program itself *unlike in 3.X*. In order to achieve the previous behavior, SceneTree.quit should be called after the notification.

Math

- Vector math
- Advanced vector math
- Matrices and transforms
- Interpolation
- Beziers, curves and paths
- Random number generation

Vector math

Introduction

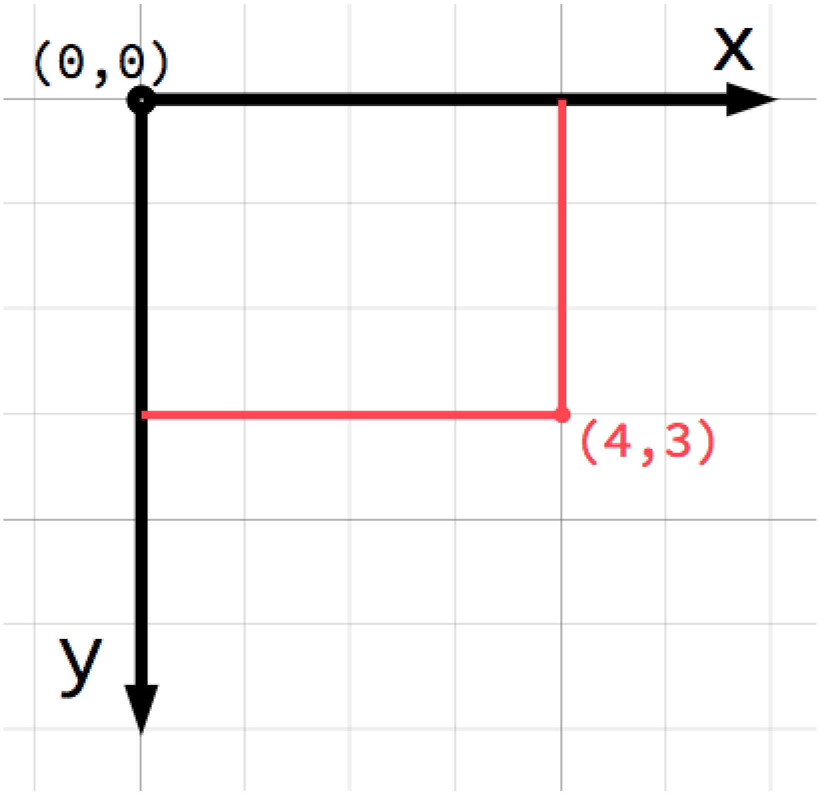
This tutorial is a short and practical introduction to linear algebra as it applies to game development. Linear algebra is the study of vectors and their uses. Vectors have many applications in both 2D and 3D development and Godot uses them extensively. Developing a good understanding of vector math is essential to becoming a strong game developer.

Note

This tutorial is **not** a formal textbook on linear algebra. We will only be looking at how it is applied to game development. For a broader look at the mathematics, see https://www.khanacademy.org/math/linear-algebra

Coordinate systems (2D)

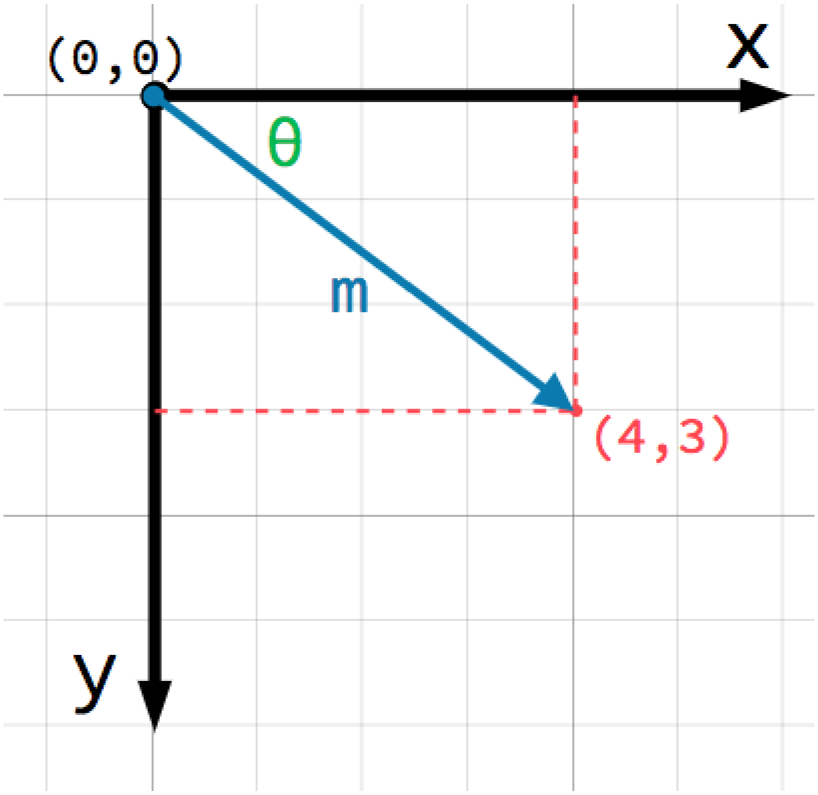
In 2D space, coordinates are defined using a horizontal axis (`x`) and a vertical axis (`y`). A particular position in 2D space is written as a pair of values such as `(4, 3)`.



Note

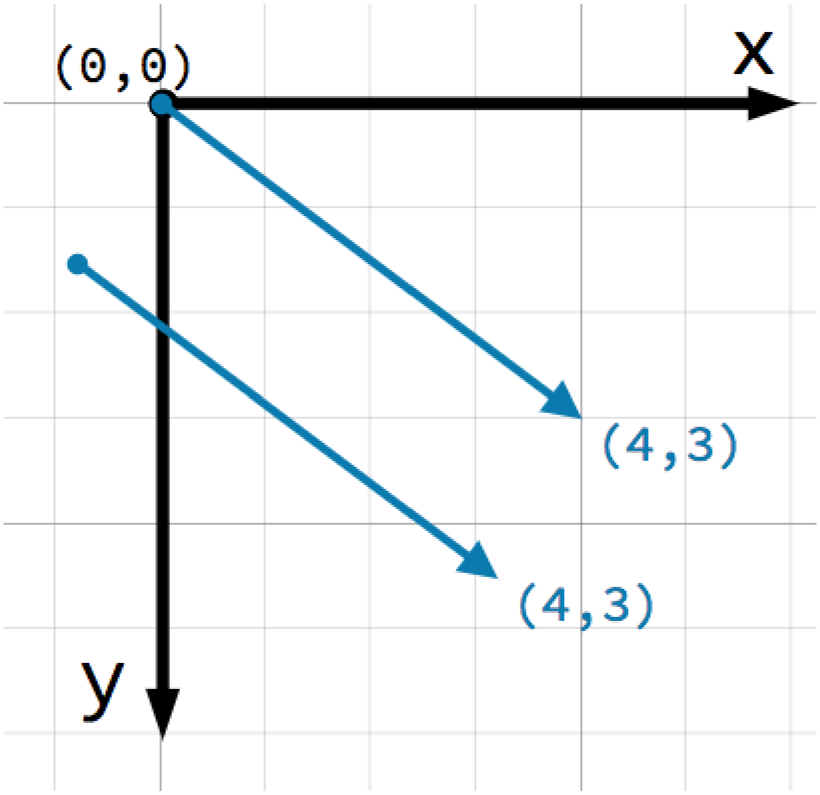
If you're new to computer graphics, it might seem odd that the positive `y` axis points **downwards** instead of upwards, as you probably learned in math class. However, this is common in most computer graphics applications.

Any position in the 2D plane can be identified by a pair of numbers in this way. However, we can also think of the position `(4, 3)` as an **offset** from the `(0, 0)` point, or **origin**. Draw an arrow pointing from the origin to the point:



This is a **vector**. A vector represents a lot of useful information. As well as telling us that the point is at `(4, 3)`, we can also think of it as an angle `θ` (theta) and a length (or magnitude) `m`. In this case, the arrow is a **position vector** - it denotes a position in space, relative to the origin.

A very important point to consider about vectors is that they only represent **relative** direction and magnitude. There is no concept of a vector's position. The following two vectors are identical:



Both vectors represent a point 4 units to the right and 3 units below some starting point. It does not matter where on the plane you draw the vector, it always represents a relative direction and magnitude.

Vector operations

You can use either method (x and y coordinates or angle and magnitude) to refer to a vector, but for convenience, programmers typically use the coordinate notation. For example, in Godot, the origin is the top-left corner of the screen, so to place a 2D node named `Node2D` 400 pixels to the right and 300 pixels down, use the following code:

GDScript

```
$Node2D.position = Vector2(400, 300)
```

C#

```
var node2D = GetNode<Node2D>("Node2D");
node2D.Position = new Vector2(400, 300);
```

Godot supports both Vector2 and Vector3 for 2D and 3D usage, respectively. The same mathematical rules discussed in this article apply to both types, and wherever we link to `Vector2` methods in the class reference, you can also check out their `Vector3` counterparts.

Member access

The individual components of the vector can be accessed directly by name.

GDScript

```
# Create a vector with coordinates (2, 5).
var a = Vector2(2, 5)
# Create a vector and assign x and y manually.
var b = Vector2()
b.x = 3
b.y = 1
```

C#

```
// Create a vector with coordinates (2, 5).
var a = new Vector2(2, 5);
// Create a vector and assign x and y manually.
var b = new Vector2();
b.X = 3;
b.Y = 1;
```

Adding vectors

When adding or subtracting two vectors, the corresponding components are added:

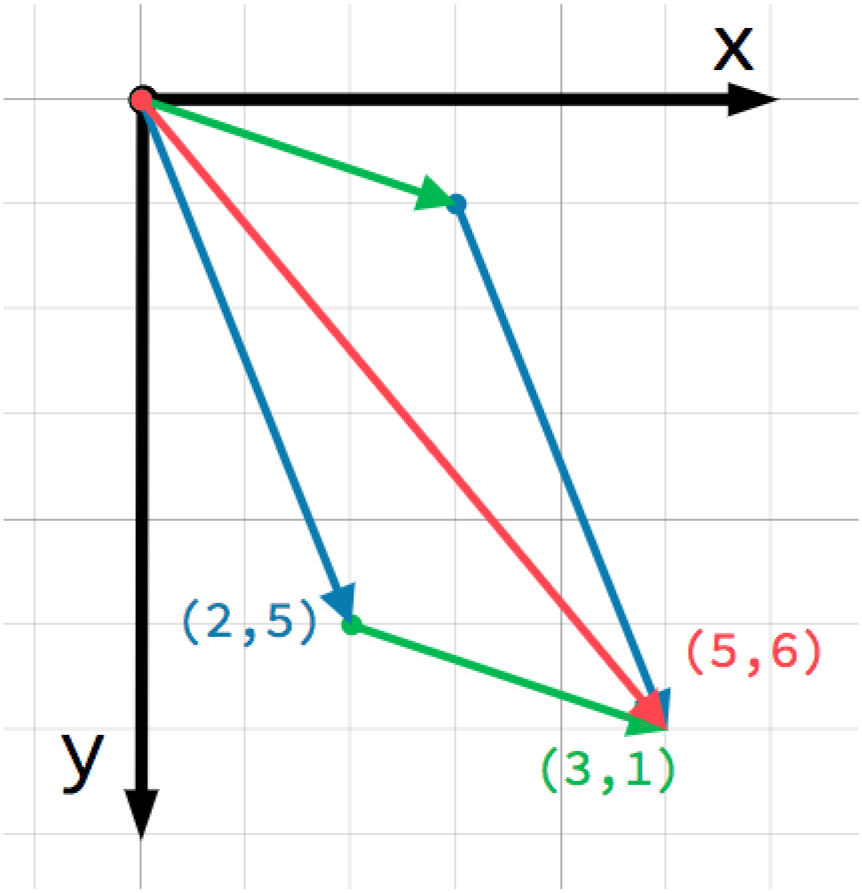
GDScript

```
var c = a + b  # (2, 5) + (3, 1) = (5, 6)
```

C#

```
var c = a + b;  // (2, 5) + (3, 1) = (5, 6)
```

We can also see this visually by adding the second vector at the end of the first:



Note that adding `a + b` gives the same result as `b + a`.

Scalar multiplication

Note

Vectors represent both direction and magnitude. A value representing only magnitude is called a **scalar**. Scalars use the float type in Godot.

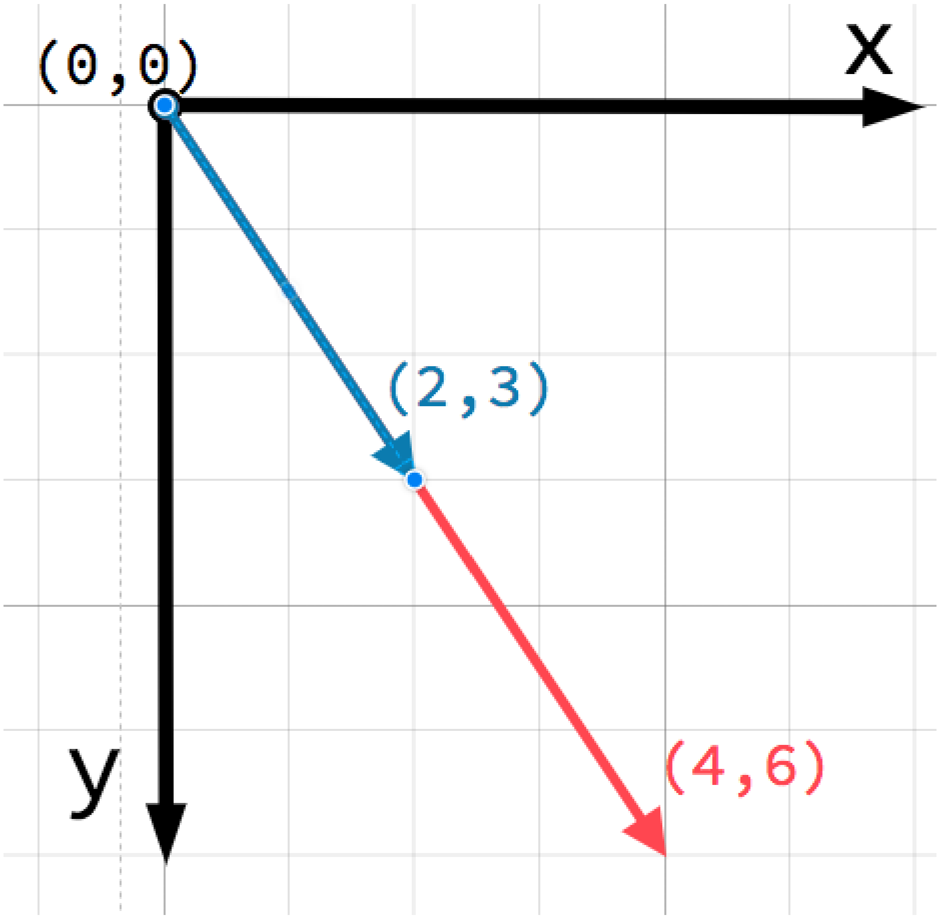
A vector can be multiplied by a **scalar**:

GDScript

```
var c = a * 2  # (2, 5) * 2 = (4, 10)
var d = b / 3  # (3, 6) / 3 = (1, 2)
var e = d * -2 # (1, 2) * -2 = (-2, -4)
```

C#

```
var c = a * 2;  // (2, 5) * 2 = (4, 10)
var d = b / 3;  // (3, 6) / 3 = (1, 2)
var e = d * -2; // (1, 2) * -2 = (-2, -4)
```



Note

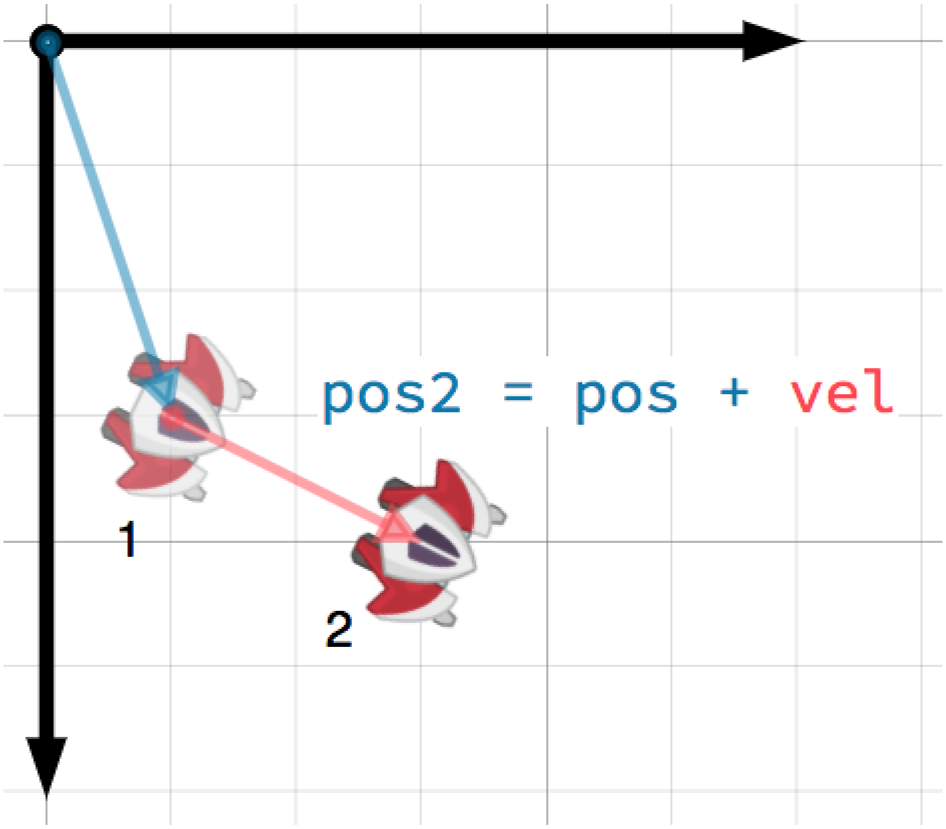
Multiplying a vector by a positive scalar does not change its direction, only its magnitude. Multiplying with a negative scalar results in a vector in the opposite direction. This is how you **scale** a vector.

Practical applications

Let's look at two common uses for vector addition and subtraction.

Movement

A vector can represent **any** quantity with a magnitude and direction. Typical examples are: position, velocity, acceleration, and force. In this image, the spaceship at step 1 has a position vector of `(1, 3)` and a velocity vector of `(2, 1)`. The velocity vector represents how far the ship moves each step. We can find the position for step 2 by adding the velocity to the current position.



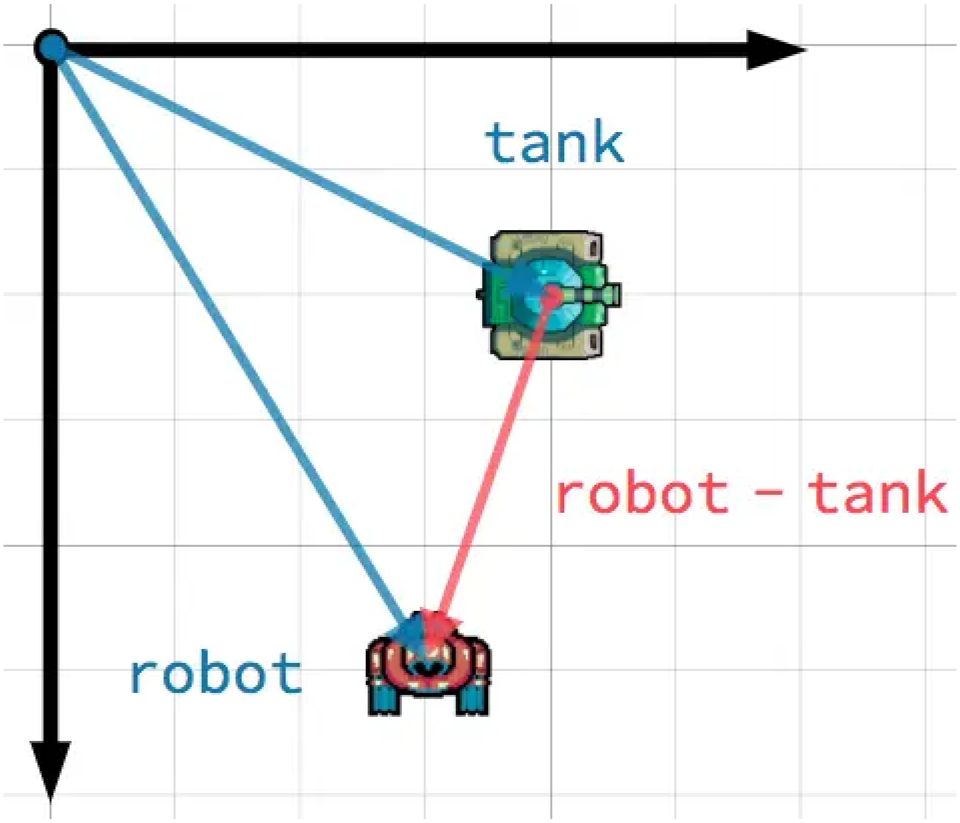
Tip

Velocity measures the **change** in position per unit of time. The new position is found by adding the velocity multiplied by the elapsed time (here assumed to be one unit, e.g. 1 s) to the previous position.

In a typical 2D game scenario, you would have a velocity in pixels per second, and multiply it by the `delta` parameter (time elapsed since the previous frame) from the _process() or _physics_process() callbacks.

Pointing toward a target

In this scenario, you have a tank that wishes to point its turret at a robot. Subtracting the tank's position from the robot's position gives the vector pointing from the tank to the robot.



Tip

To find a vector pointing from `A` to `B`, use `B - A`.

Unit vectors

A vector with **magnitude** of `1` is called a **unit vector**. They are also sometimes referred to as **direction vectors** or **normals**. Unit vectors are helpful when you need to keep track of a direction.

Normalization

Normalizing a vector means reducing its length to `1` while preserving its direction. This is done by dividing each of its components by its magnitude. Because this is such a common operation, Godot provides a dedicated normalized() method for this:

GDScript

```
a = a.normalized()
```

C#

```
a = a.Normalized();
```

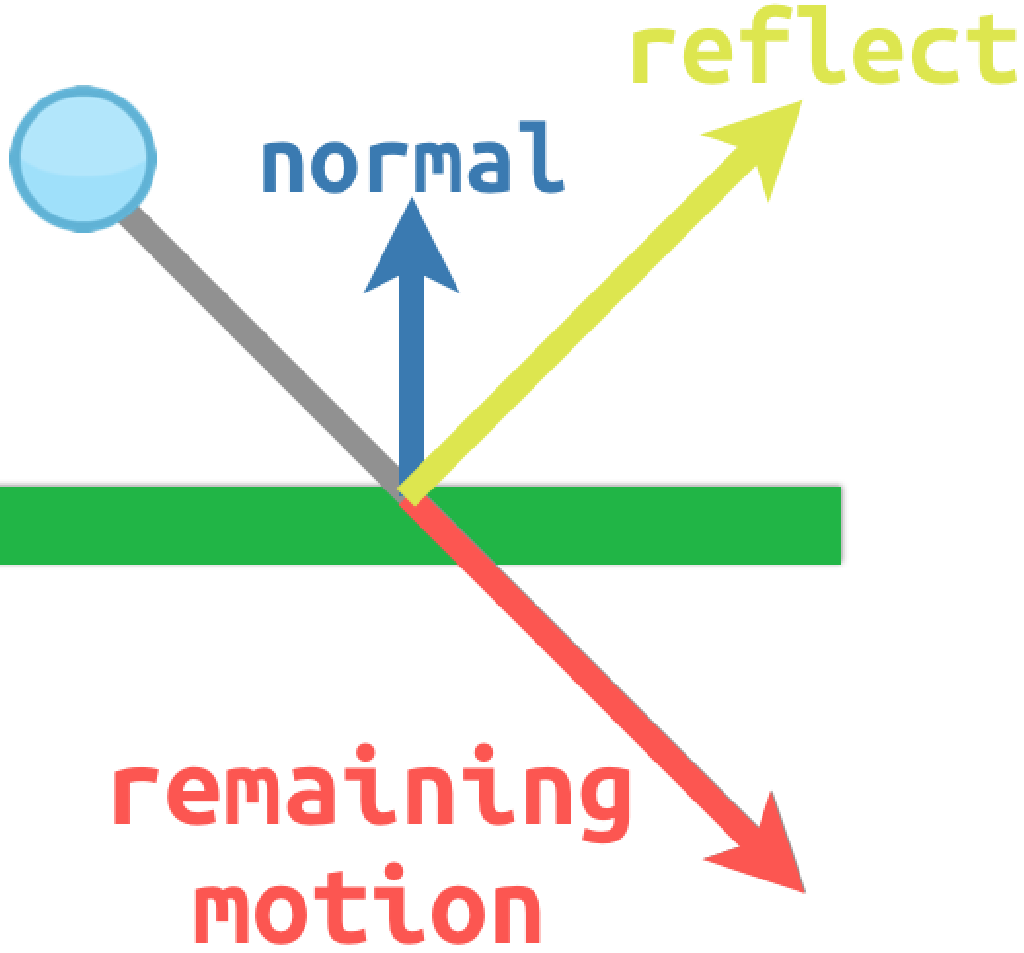
Warning

Because normalization involves dividing by the vector's length, you cannot normalize a vector of length `0`. Attempting to do so would normally result in an error. In GDScript though, trying to call the `normalized()` method on a vector of length 0 leaves the value untouched and avoids the error for you.

Reflection

A common use of unit vectors is to indicate **normals**. Normal vectors are unit vectors aligned perpendicularly to a surface, defining its direction. They are commonly used for lighting, collisions, and other operations involving surfaces.

For example, imagine we have a moving ball that we want to bounce off a wall or other object:



The surface normal has a value of `(0, -1)` because this is a horizontal surface. When the ball collides, we take its remaining motion (the amount left over when it hits the surface) and reflect it using the normal. In Godot, there is a bounce() method to handle this. Here is a code example of the above diagram using a CharacterBody2D:

GDScript

```
var collision: KinematicCollision2D = move_and_collide(velocity
* delta)
if collision:
    var reflect =
collision.get_remainder().bounce(collision.get_normal())
    velocity = velocity.bounce(collision.get_normal())
    move_and_collide(reflect)
```

C#

```
KinematicCollision2D collision = MoveAndCollide(_velocity *
(float)delta);
if (collision != null)
{
    var reflect =
collision.GetRemainder().Bounce(collision.GetNormal());
    _velocity = _velocity.Bounce(collision.GetNormal());
    MoveAndCollide(reflect);
}
```

Dot product

The **dot product** is one of the most important concepts in vector math, but is often misunderstood. Dot product is an operation on two vectors that returns a **scalar**. Unlike a vector, which contains both magnitude and direction, a scalar value has only magnitude.

The formula for dot product takes two common forms:

$$A \cdot B = ||A||\ ||B|| \cos \Theta$$

and

$$A \cdot B = A_x B_x + A_y B_y$$

The mathematical notation *||A||* represents the magnitude of vector `A`, and A_x means the `x` component of vector `A`.

However, in most cases it is easiest to use the built-in dot() method. Note that the order of the two vectors does not matter:

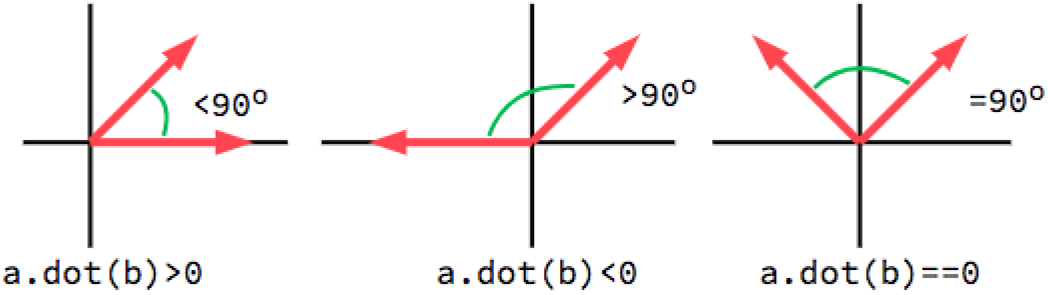
GDScript

```
var c = a.dot(b)
var d = b.dot(a)   # These are equivalent.
```

C#

```
float c = a.Dot(b);
float d = b.Dot(a);   // These are equivalent.
```

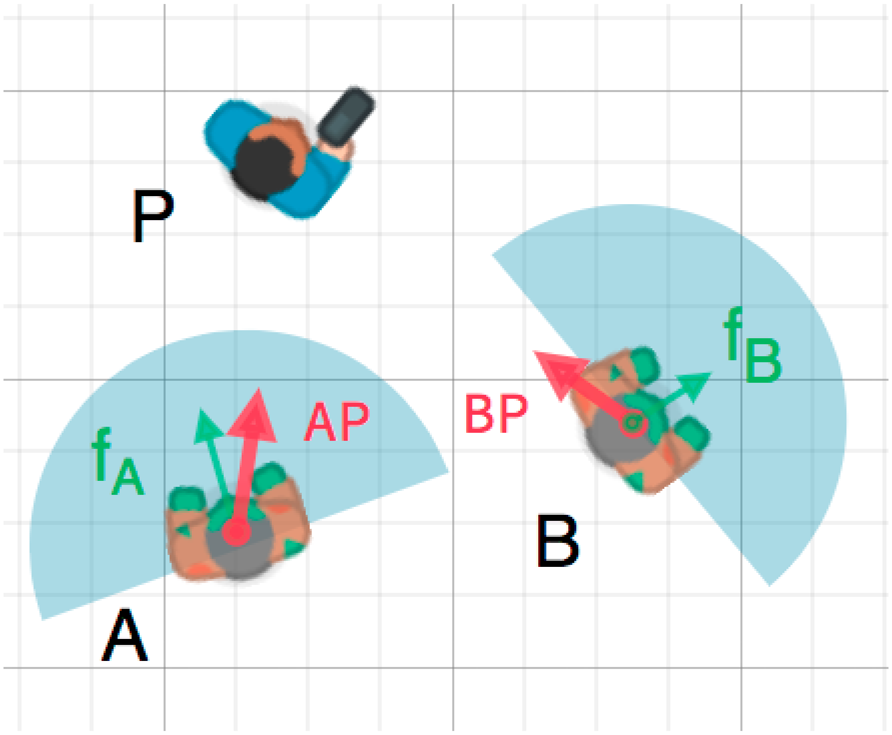
The dot product is most useful when used with unit vectors, making the first formula reduce to just `cos(θ)`. This means we can use the dot product to tell us something about the angle between two vectors:



When using unit vectors, the result will always be between `-1` (180°) and `1` (0°).

Facing

We can use this fact to detect whether an object is facing toward another object. In the diagram below, the player `P` is trying to avoid the zombies `A` and `B`. Assuming a zombie's field of view is **180°**, can they see the player?



The green arrows `fA` and `fB` are **unit vectors** representing the zombie's facing direction and the blue semicircle represents its field of view. For zombie `A`, we find the direction vector `AP` pointing to the player using `P - A` and normalize it, however, Godot has a helper method to do this called direction_to(). If the angle between this vector and the facing vector is less than 90°, then the zombie can see the player.

In code it would look like this:

GDScript

```
var AP = A.direction_to(P)
if AP.dot(fA) > 0:
    print("A sees P!")
```

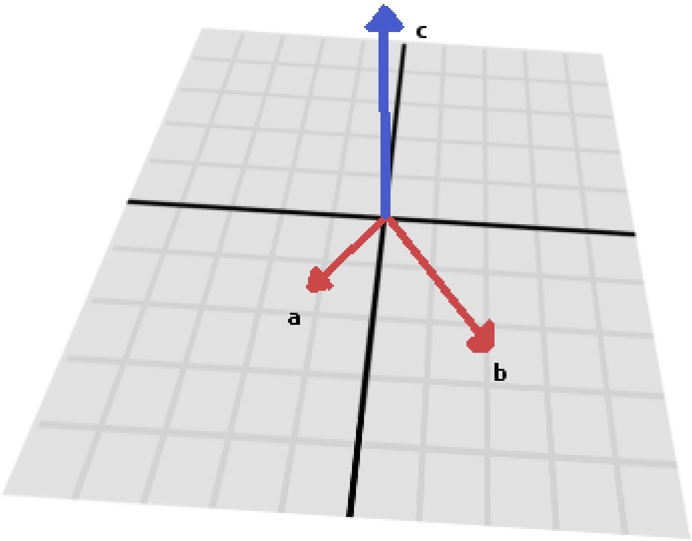
C#

```
var AP = A.DirectionTo(P);
if (AP.Dot(fA) > 0)
{
    GD.Print("A sees P!");
}
```

Cross product

Like the dot product, the **cross product** is an operation on two vectors. However, the result of the cross product is a vector with a direction that is perpendicular to both. Its magnitude depends on their relative angle. If two vectors are parallel, the result of their cross product will be a null vector.

$$||a \times b|| = ||a||\ ||b||\ \ |\sin(a, b)|$$



The cross product is calculated like this:

GDScript

```
var c = Vector3()
c.x = (a.y * b.z) - (a.z * b.y)
c.y = (a.z * b.x) - (a.x * b.z)
c.z = (a.x * b.y) - (a.y * b.x)
```

C#

```
var c = new Vector3();
c.X = (a.Y * b.Z) - (a.Z * b.Y);
c.Y = (a.Z * b.X) - (a.X * b.Z);
c.Z = (a.X * b.Y) - (a.Y * b.X);
```

With Godot, you can use the built-in Vector3.cross() method:

GDScript

```
var c = a.cross(b)
```

C#

```
var c = a.Cross(b);
```

The cross product is not mathematically defined in 2D. The Vector2.cross() method is a commonly used analog of the 3D cross product for 2D vectors.

Note

In the cross product, order matters. `a.cross(b)` does not give the same result as `b.cross(a)`. The resulting vectors point in **opposite** directions.

Calculating normals

One common use of cross products is to find the surface normal of a plane or surface in 3D space. If we have the triangle `ABC` we can use vector subtraction to find two edges `AB` and `AC`. Using the cross product, `AB × AC` produces a vector perpendicular to both: the surface normal.

Here is a function to calculate a triangle's normal:

GDScript

```
func get_triangle_normal(a, b, c):
    # Find the surface normal given 3 vertices.
    var side1 = b - a
    var side2 = c - a
    var normal = side1.cross(side2)
    return normal
```

C#

```
Vector3 GetTriangleNormal(Vector3 a, Vector3 b, Vector3 c)
{
    // Find the surface normal given 3 vertices.
    var side1 = b - a;
    var side2 = c - a;
    var normal = side1.Cross(side2);
    return normal;
}
```

Pointing to a target

In the dot product section above, we saw how it could be used to find the angle between two vectors. However, in 3D, this is not enough information. We also need to know what axis to rotate around. We can find that by calculating the cross product of the current facing direction and the target direction. The resulting perpendicular vector is the axis of rotation.

More information

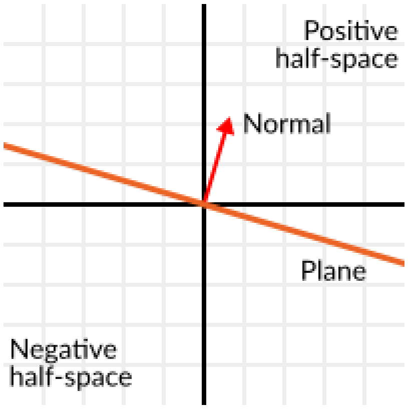
For more information on using vector math in Godot, see the following articles:

- Advanced vector math
- Matrices and transforms

Advanced vector math

Planes

The dot product has another interesting property with unit vectors. Imagine that perpendicular to that vector (and through the origin) passes a plane. Planes divide the entire space into positive (over the plane) and negative (under the plane), and (contrary to popular belief) you can also use their math in 2D:



Unit vectors that are perpendicular to a surface (so, they describe the orientation of the surface) are called **unit normal vectors**. Though, usually they are just abbreviated as *normals*. Normals appear in planes, 3D geometry (to determine where each face or vertex is siding), etc. A **normal** *is* a **unit vector**, but it's called *normal* because of its usage. (Just like we call (0,0) the Origin!).

The plane passes by the origin and the surface of it is perpendicular to the unit vector (or *normal*). The side towards the vector points to is the positive half-space, while the other side is the negative half-space. In 3D this is exactly the same, except that the plane is an infinite surface (imagine an infinite, flat sheet of paper that you can orient and is pinned to the origin) instead of a line.

Distance to plane

Now that it's clear what a plane is, let's go back to the dot product. The dot product between a **unit vector** and any **point in space** (yes, this time we do dot product between vector and position), returns the **distance from the point to the plane**:

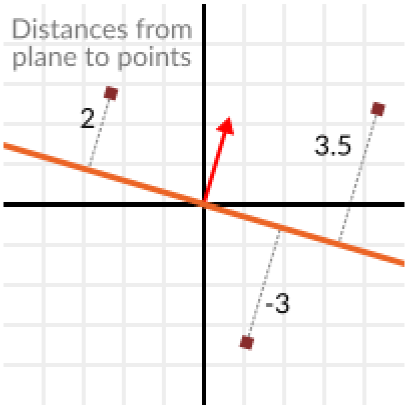
GDScript

```
var distance = normal.dot(point)
```

C#

```
var distance = normal.Dot(point);
```

But not just the absolute distance, if the point is in the negative half space the distance will be negative, too:



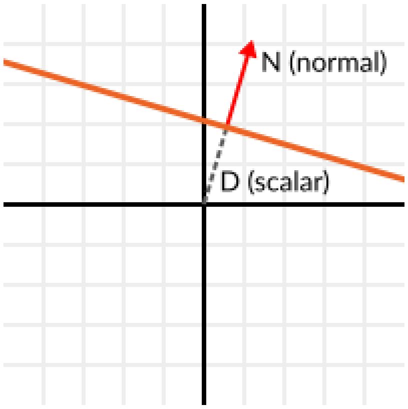
This allows us to tell which side of the plane a point is.

Away from the origin

I know what you are thinking! So far this is nice, but *real* planes are everywhere in space, not only passing through the origin. You want real *plane* action and you want it *now*.

Remember that planes not only split space in two, but they also have *polarity*. This means that it is possible to have perfectly overlapping planes, but their negative and positive half-spaces are swapped.

With this in mind, let's describe a full plane as a **normal** *N* and a **distance from the origin** scalar *D*. Thus, our plane is represented by N and D. For example:



For 3D math, Godot provides a Plane built-in type that handles this.

Basically, N and D can represent any plane in space, be it for 2D or 3D (depending on the amount of dimensions of N) and the math is the same for both. It's the same as before, but D is the distance from the origin to the plane, travelling in N direction. As an example, imagine you want to reach a point in the plane, you will just do:

GDScript

```
var point_in_plane = N*D
```

C#

```
var pointInPlane = N * D;
```

This will stretch (resize) the normal vector and make it touch the plane. This math might seem confusing, but it's actually much simpler than it seems. If we want to tell, again, the distance from the point to the plane, we do the same but adjusting for distance:

GDScript

```
var distance = N.dot(point) - D
```

C#

```
var distance = N.Dot(point) - D;
```

The same thing, using a built-in function:

GDScript

```
var distance = plane.distance_to(point)
```

C#

```
var distance = plane.DistanceTo(point);
```

This will, again, return either a positive or negative distance.

Flipping the polarity of the plane can be done by negating both N and D. This will result in a plane in the same position, but with inverted negative and positive half spaces:

GDScript

```
N = -N
D = -D
```

C#

```
N = -N;
D = -D;
```

Godot also implements this operator in Plane. So, using the format below will work as expected:

GDScript

```
var inverted_plane = -plane
```

C#

```
var invertedPlane = -plane;
```

So, remember, the plane's main practical use is that we can calculate the distance to it. So, when is it useful to calculate the distance from a point to a plane? Let's see some examples.

Constructing a plane in 2D

Planes clearly don't come out of nowhere, so they must be built. Constructing them in 2D is easy, this can be done from either a normal (unit vector) and a point, or from two points in space.

In the case of a normal and a point, most of the work is done, as the normal is already computed, so calculate D from the dot product of the normal and the point.

GDScript

```
var N = normal
var D = normal.dot(point)
```

C#

```
var N = normal;
var D = normal.Dot(point);
```

For two points in space, there are actually two planes that pass through them, sharing the same space but with normal pointing to the opposite directions. To compute the normal from the two points, the direction vector must be obtained first, and then it needs to be rotated 90 degrees to either side:

GDScript

```
# Calculate vector from `a` to `b`.
var dvec = point_a.direction_to(point_b)
# Rotate 90 degrees.
var normal = Vector2(dvec.y, -dvec.x)
# Alternatively (depending the desired side of the normal):
# var normal = Vector2(-dvec.y, dvec.x)
```

C#

```
// Calculate vector from `a` to `b`.
var dvec = pointA.DirectionTo(pointB);
// Rotate 90 degrees.
var normal = new Vector2(dvec.Y, -dvec.X);
// Alternatively (depending the desired side of the normal):
// var normal = new Vector2(-dvec.Y, dvec.X);
```

The rest is the same as the previous example. Either point_a or point_b will work, as they are in the same plane:

GDScript

```
var N = normal
var D = normal.dot(point_a)
# this works the same
# var D = normal.dot(point_b)
```

C#

```
var N = normal;
var D = normal.Dot(pointA);
// this works the same
// var D = normal.Dot(pointB);
```

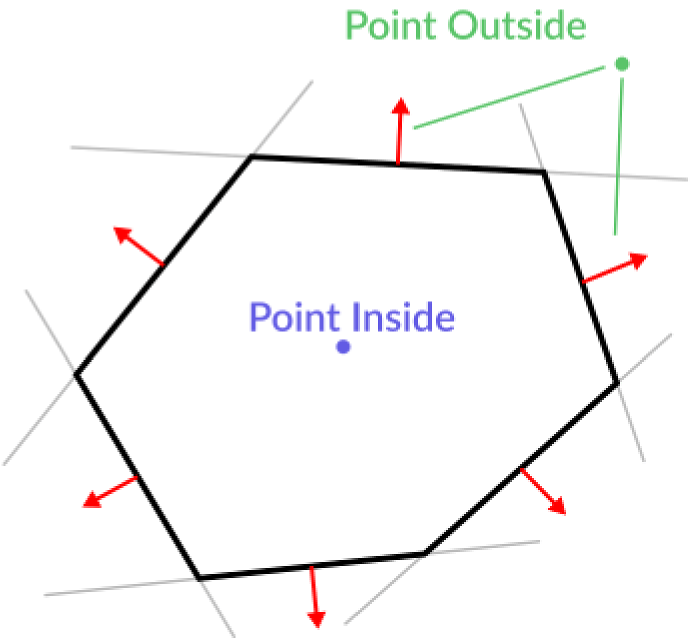
Doing the same in 3D is a little more complex and is explained further down.

Some examples of planes

Here is an example of what planes are useful for. Imagine you have a convex [https://www.mathsisfun.com/definitions/convex.html] polygon. For example, a rectangle, a trapezoid, a triangle, or just any polygon where no faces bend inwards.

For every segment of the polygon, we compute the plane that passes by that segment. Once we have the list of planes, we can do neat things, for example checking if a point is inside the polygon.

We go through all planes, if we can find a plane where the distance to the point is positive, then the point is outside the polygon. If we can't, then the point is inside.



Code should be something like this:

GDScript

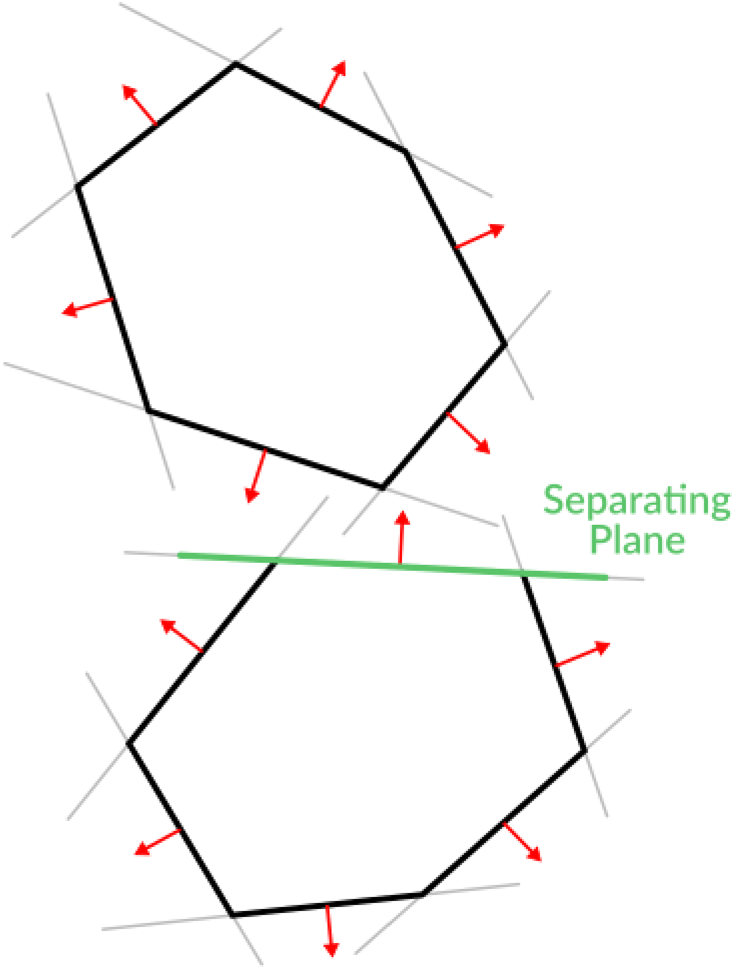
```
var inside = true
for p in planes:
    # check if distance to plane is positive
    if (p.distance_to(point) > 0):
        inside = false
        break # with one that fails, it's enough
```

C#

```
var inside = true;
foreach (var p in planes)
{
    // check if distance to plane is positive
    if (p.DistanceTo(point) > 0)
    {
        inside = false;
        break; // with one that fails, it's enough
    }
}
```

Pretty cool, huh? But this gets much better! With a little more effort, similar logic will let us know when two convex polygons are overlapping too. This is called the Separating Axis Theorem (or SAT) and most physics engines use this to detect collision.

With a point, just checking if a plane returns a positive distance is enough to tell if the point is outside. With another polygon, we must find a plane where *all the other polygon points* return a positive distance to it. This check is performed with the planes of A against the points of B, and then with the planes of B against the points of A:



Code should be something like this:

GDScript

```
var overlapping = true

for p in planes_of_A:
    var all_out = true
    for v in points_of_B:
        if (p.distance_to(v) < 0):
            all_out = false
            break

    if (all_out):
        # a separating plane was found
        # do not continue testing
        overlapping = false
        break

if (overlapping):
    # only do this check if no separating plane
    # was found in planes of A
    for p in planes_of_B:
        var all_out = true
        for v in points_of_A:
            if (p.distance_to(v) < 0):
                all_out = false
                break

        if (all_out):
            overlapping = false
            break

if (overlapping):
    print("Polygons Collided!")
```

C#

```
var overlapping = true;

foreach (Plane plane in planesOfA)
{
    var allOut = true;
    foreach (Vector3 point in pointsOfB)
    {
        if (plane.DistanceTo(point) < 0)
        {
            allOut = false;
            break;
        }
    }

    if (allOut)
    {
        // a separating plane was found
        // do not continue testing
        overlapping = false;
        break;
    }
}

if (overlapping)
{
    // only do this check if no separating plane
    // was found in planes of A
    foreach (Plane plane in planesOfB)
    {
        var allOut = true;
        foreach (Vector3 point in pointsOfA)
        {
            if (plane.DistanceTo(point) < 0)
            {
                allOut = false;
                break;
            }
        }

        if (allOut)
        {
            overlapping = false;
            break;
        }
    }
}

if (overlapping)
{
    GD.Print("Polygons Collided!");
}
```

As you can see, planes are quite useful, and this is the tip of the iceberg. You might be wondering what happens with non convex polygons. This is usually just handled by splitting the concave polygon into smaller convex polygons, or using a technique such as BSP (which is not used much nowadays).

Collision detection in 3D

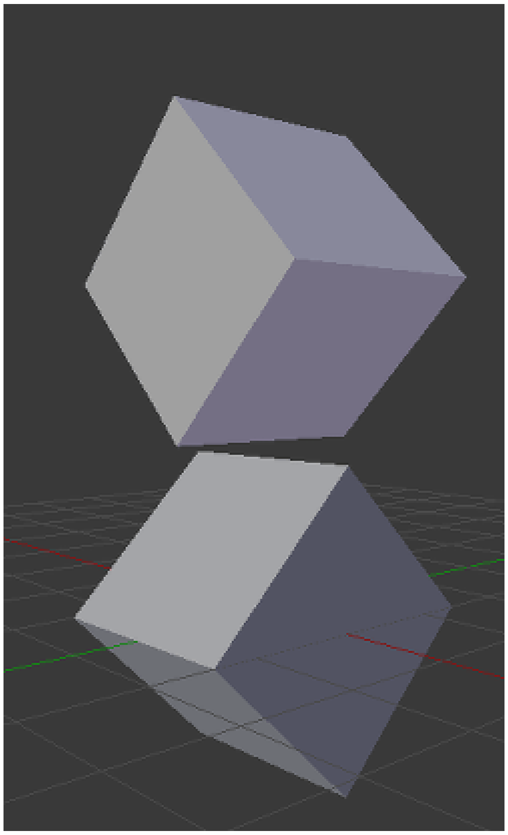
This is another bonus bit, a reward for being patient and keeping up with this long tutorial. Here is another piece of wisdom. This might not be something with a direct use case (Godot already does collision detection pretty well) but it's used by almost all physics engines and collision detection libraries :)

Remember that converting a convex shape in 2D to an array of 2D planes was useful for collision detection? You could detect if a point was inside any convex shape, or if two 2D convex shapes were overlapping.

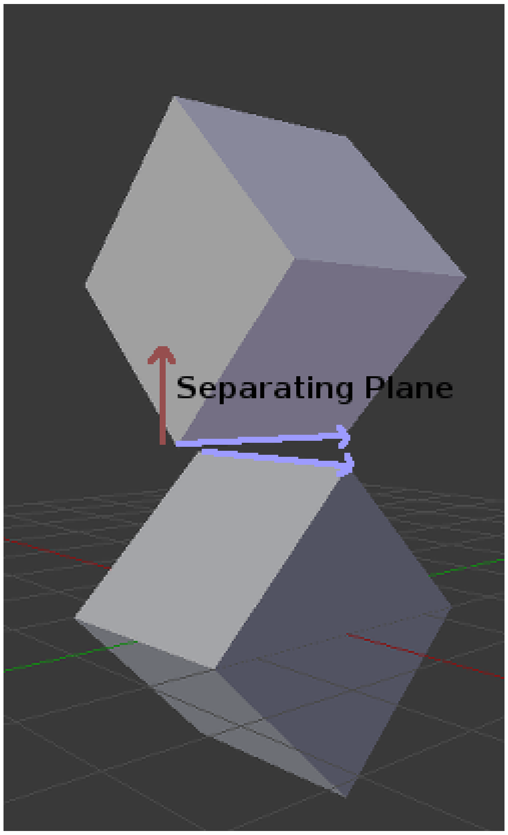
Well, this works in 3D too, if two 3D polyhedral shapes are colliding, you won't be able to find a separating plane. If a separating plane is found, then the shapes are definitely not colliding.

To refresh a bit a separating plane means that all vertices of polygon A are in one side of the plane, and all vertices of polygon B are in the other side. This plane is always one of the face-planes of either polygon A or polygon B.

In 3D though, there is a problem to this approach, because it is possible that, in some cases a separating plane can't be found. This is an example of such situation:



To avoid it, some extra planes need to be tested as separators, these planes are the cross product between the edges of polygon A and the edges of polygon B



So the final algorithm is something like:

GDScript

```
var overlapping = true

for p in planes_of_A:
    var all_out = true
    for v in points_of_B:
        if (p.distance_to(v) < 0):
            all_out = false
            break

    if (all_out):
        # a separating plane was found
        # do not continue testing
        overlapping = false
        break

if (overlapping):
    # only do this check if no separating plane
    # was found in planes of A
    for p in planes_of_B:
        var all_out = true
        for v in points_of_A:
            if (p.distance_to(v) < 0):
                all_out = false
                break

        if (all_out):
            overlapping = false
            break

if (overlapping):
    for ea in edges_of_A:
        for eb in edges_of_B:
            var n = ea.cross(eb)
            if (n.length() == 0):
                continue

            var max_A = -1e20 # tiny number
            var min_A = 1e20 # huge number

            # we are using the dot product directly
            # so we can map a maximum and minimum range
            # for each polygon, then check if they
            # overlap.

            for v in points_of_A:
                var d = n.dot(v)
                max_A = max(max_A, d)
                min_A = min(min_A, d)

            var max_B = -1e20 # tiny number
            var min_B = 1e20 # huge number

            for v in points_of_B:
                var d = n.dot(v)
                max_B = max(max_B, d)
                min_B = min(min_B, d)

            if (min_A > max_B or min_B > max_A):
                # not overlapping!
                overlapping = false
                break

        if (not overlapping):
            break

if (overlapping):
   print("Polygons collided!")
```

C#

```
var overlapping = true;

foreach (Plane plane in planesOfA)
{
    var allOut = true;
    foreach (Vector3 point in pointsOfB)
    {
        if (plane.DistanceTo(point) < 0)
        {
            allOut = false;
            break;
        }
    }

    if (allOut)
    {
        // a separating plane was found
        // do not continue testing
        overlapping = false;
        break;
    }
}

if (overlapping)
{
    // only do this check if no separating plane
    // was found in planes of A
    foreach (Plane plane in planesOfB)
    {
        var allOut = true;
        foreach (Vector3 point in pointsOfA)
        {
            if (plane.DistanceTo(point) < 0)
            {
                allOut = false;
                break;
            }
        }

        if (allOut)
        {
            overlapping = false;
            break;
        }
    }
}

if (overlapping)
{
    foreach (Vector3 edgeA in edgesOfA)
    {
        foreach (Vector3 edgeB in edgesOfB)
        {
            var normal = edgeA.Cross(edgeB);
            if (normal.Length() == 0)
            {
                continue;
            }

            var maxA = float.MinValue; // tiny number
            var minA = float.MaxValue; // huge number

            // we are using the dot product directly
            // so we can map a maximum and minimum range
            // for each polygon, then check if they
            // overlap.

            foreach (Vector3 point in pointsOfA)
            {
                var distance = normal.Dot(point);
                maxA = Mathf.Max(maxA, distance);
                minA = Mathf.Min(minA, distance);
            }
            var maxB = float.MinValue; // tiny number
            var minB = float.MaxValue; // huge number

            foreach (Vector3 point in pointsOfB)
            {
                var distance = normal.Dot(point);
                maxB = Mathf.Max(maxB, distance);
                minB = Mathf.Min(minB, distance);
            }

            if (minA > maxB || minB > maxA)
            {
                // not overlapping!
                overlapping = false;
                break;
            }
        }

        if (!overlapping)
        {
            break;
        }

    }
}

if (overlapping)
{
    GD.Print("Polygons Collided!");
}
```

More information

For more information on using vector math in Godot, see the following article:

- Matrices and transforms

If you would like additional explanation, you should check out 3Blue1Brown's excellent video series "Essence of Linear Algebra": https://www.youtube.com/watch?v=fNk_zzaMoSs&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab

Matrices and transforms

Introduction

Before reading this tutorial, we recommend that you thoroughly read and understand the Vector math tutorial, as this tutorial requires a knowledge of vectors.

This tutorial is about *transformations* and how we represent them in Godot using matrices. It is not a full in-depth guide to matrices. Transformations are most of the time applied as translation, rotation, and scale, so we will focus on how to represent those with matrices.

Most of this guide focuses on 2D, using Transform2D and Vector2, but the way things work in 3D is very similar.

Note

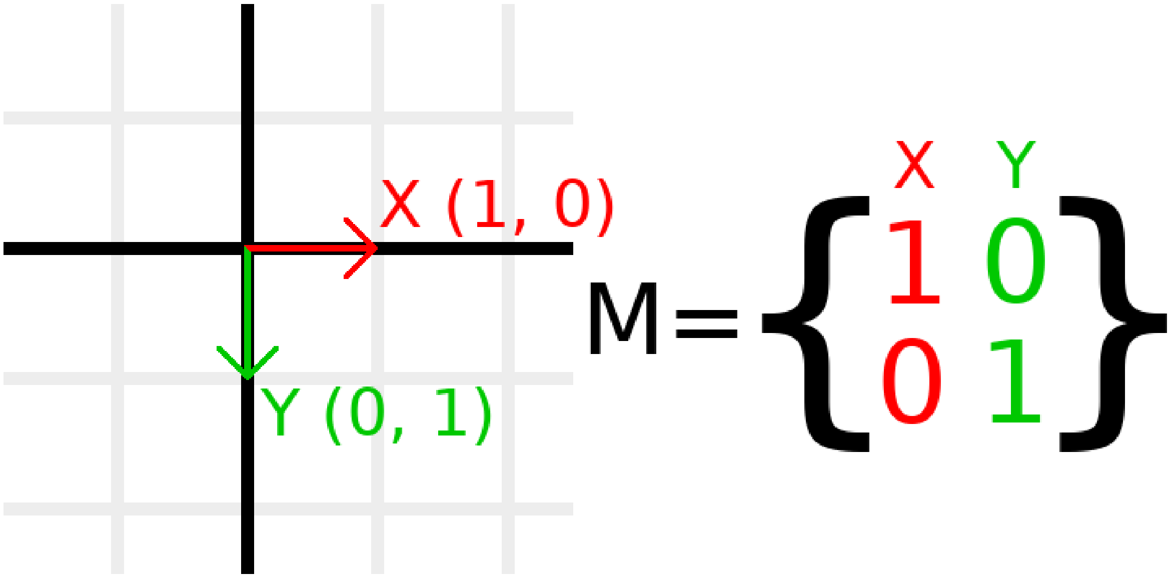
As mentioned in the previous tutorial, it is important to remember that in Godot, the Y axis points *down* in 2D. This is the opposite of how most schools teach linear algebra, with the Y axis pointing up.

Note

The convention is that the X axis is red, the Y axis is green, and the Z axis is blue. This tutorial is color-coded to match these conventions, but we will also represent the origin vector with a blue color.

Matrix components and the Identity matrix

The identity matrix represents a transform with no translation, no rotation, and no scale. Let's start by looking at the identity matrix and how its components relate to how it visually appears.



Matrices have rows and columns, and a transformation matrix has specific conventions on what each does.

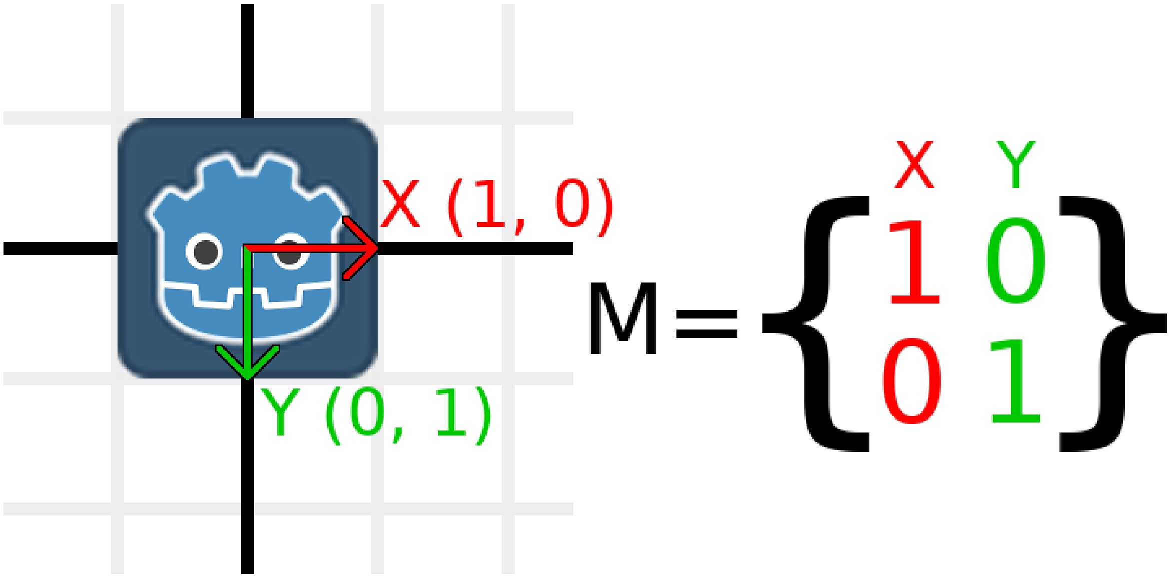
In the image above, we can see that the red X vector is represented by the first column of the matrix, and the green Y vector is likewise represented by the second column. A change to the columns will change these vectors. We will see how they can be manipulated in the next few examples.

You should not worry about manipulating rows directly, as we usually work with columns. However, you can think of the rows of the matrix as showing which vectors contribute to moving in a given direction.

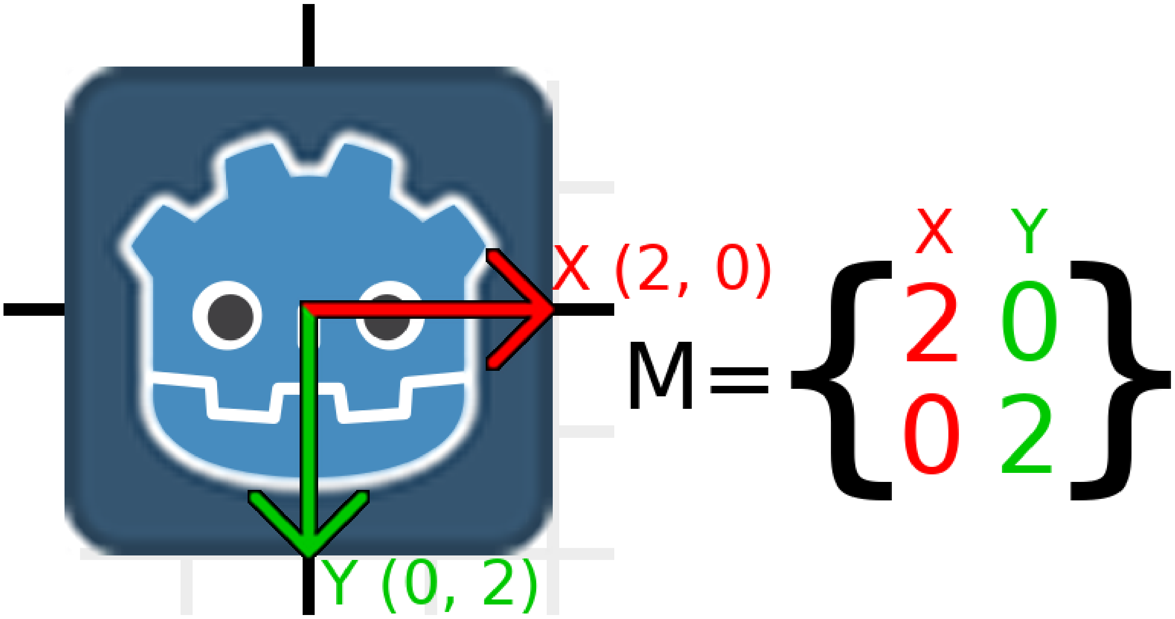
When we refer to a value such as `t.x.y`, that's the Y component of the X column vector. In other words, the bottom-left of the matrix. Similarly, `t.x.x` is top-left, `t.y.x` is top-right, and `t.y.y` is bottom-right, where `t` is the Transform2D.

Scaling the transformation matrix

Applying a scale is one of the easiest operations to understand. Let's start by placing the Godot logo underneath our vectors so that we can visually see the effects on an object:



Now, to scale the matrix, all we need to do is multiply each component by the scale we want. Let's scale it up by 2. 1 times 2 becomes 2, and 0 times 2 becomes 0, so we end up with this:



To do this in code, we multiply each of the vectors:

GDScript

```
var t = Transform2D()
# Scale
t.x *= 2
t.y *= 2
transform = t # Change the node's transform to what we
calculated.
```

C#

```
Transform2D t = Transform2D.Identity;
// Scale
t.X *= 2;
t.Y *= 2;
Transform = t; // Change the node's transform to what we
calculated.
```

If we wanted to return it to its original scale, we can multiply each component by 0.5. That's pretty much all there is to scaling a transformation matrix.

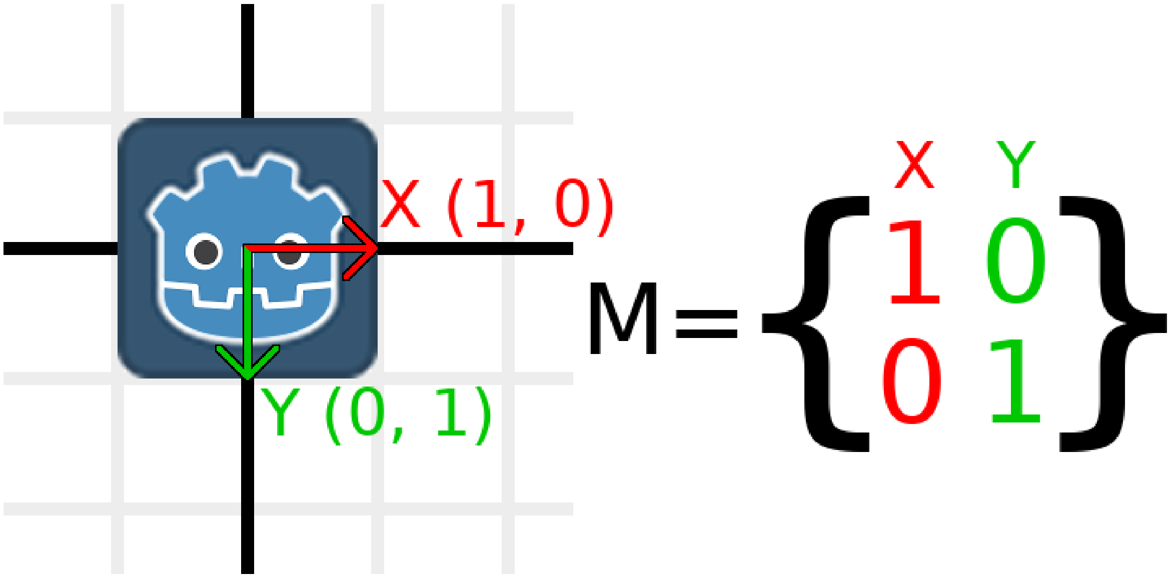
To calculate the object's scale from an existing transformation matrix, you can use `length()` on each of the column vectors.

Note

In actual projects, you can use the `scaled()` method to perform scaling.

Rotating the transformation matrix

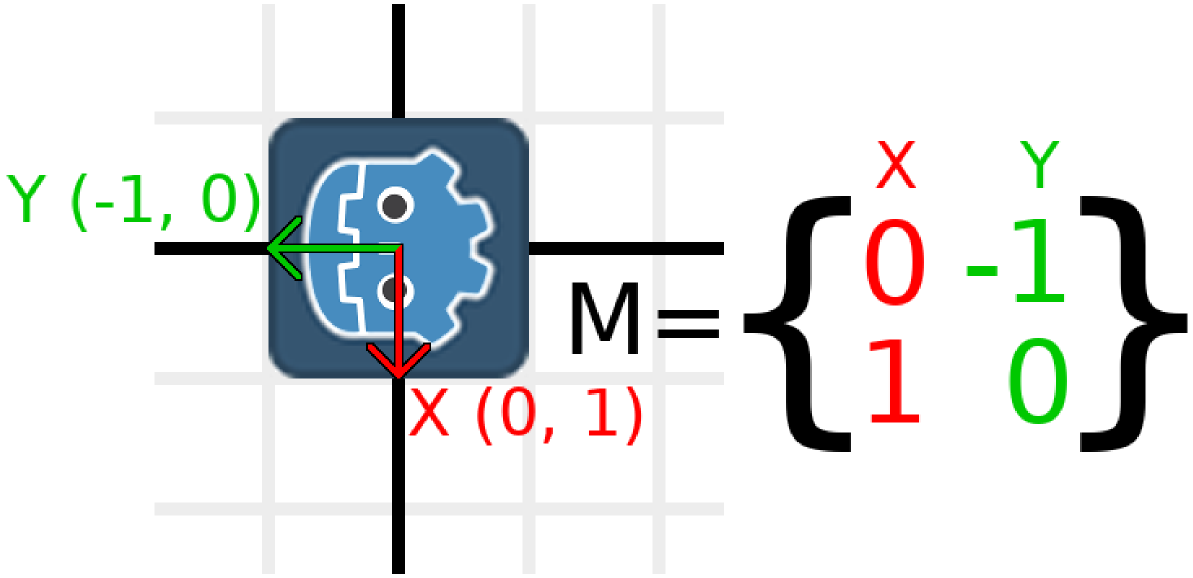
We'll start the same way as earlier, with the Godot logo underneath the identity matrix:



As an example, let's say we want to rotate our Godot logo clockwise by 90 degrees. Right now the X axis points right and the Y axis points down. If we rotate these in our head, we would logically see that the new X axis should point down and the new Y axis should point left.

You can imagine that you grab both the Godot logo and its vectors, and then spin it around the center. Wherever you finish spinning, the orientation of the vectors determines what the matrix is.

We need to represent "down" and "left" in normal coordinates, so means we'll set X to (0, 1) and Y to (-1, 0). These are also the values of `Vector2.DOWN` and `Vector2.LEFT`. When we do this, we get the desired result of rotating the object:



If you have trouble understanding the above, try this exercise: Cut a square of paper, draw X and Y vectors on top of it, place it on graph paper, then rotate it and note the endpoints.

To perform rotation in code, we need to be able to calculate the values programmatically. This image shows the formulas needed to calculate the transformation matrix from a rotation angle. Don't worry if this part seems complicated, I promise it's the hardest thing you need to know.

$$M=\left\{\begin{array}{cc} X & Y \\ \cos(rot) & -\sin(rot) \\ \sin(rot) & \cos(rot) \end{array}\right\}$$

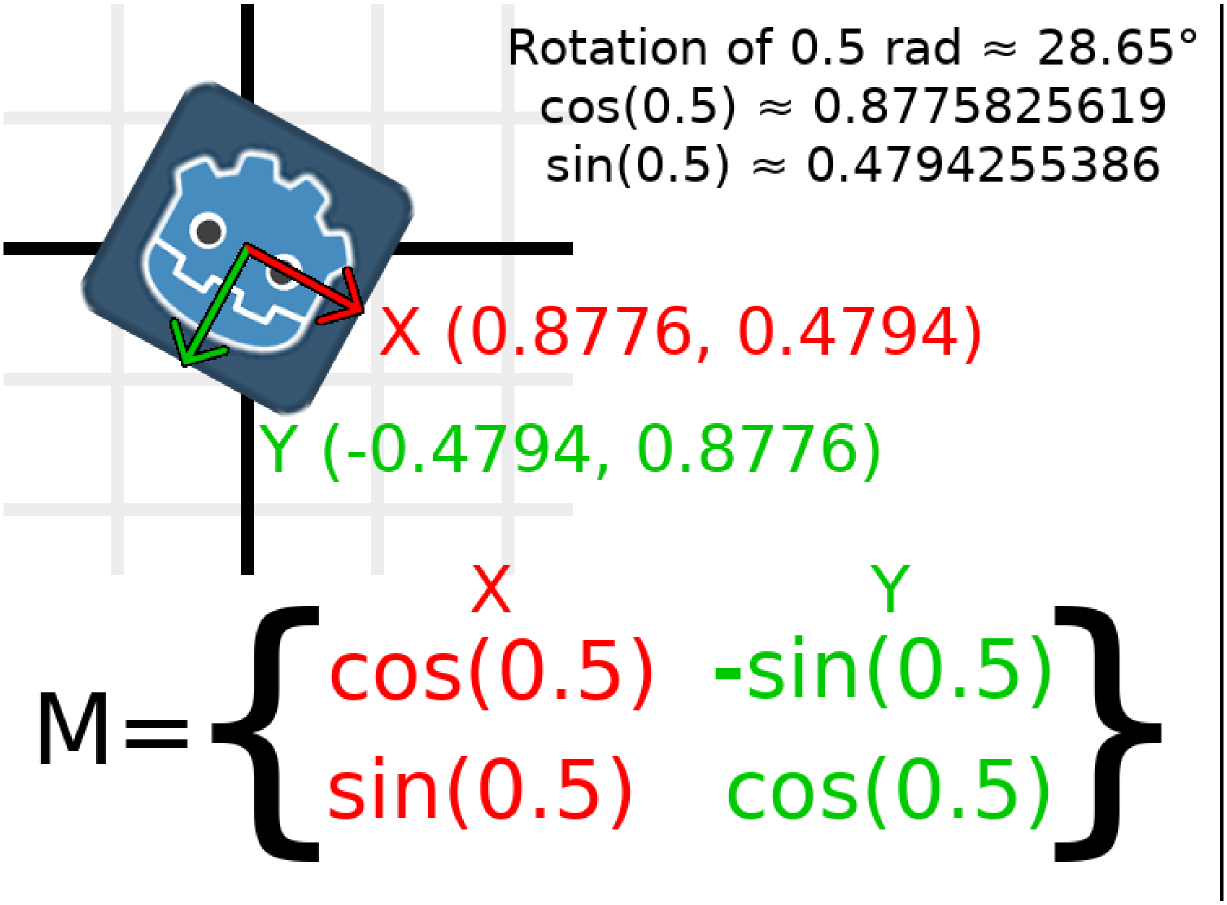
Note

Godot represents all rotations with radians, not degrees. A full turn is *TAU* or *PI*2* radians, and a quarter turn of 90 degrees is *TAU/4* or *PI/2* radians. Working with *TAU* usually results in more readable code.

Note

Fun fact: In addition to Y being *down* in Godot, rotation is represented clockwise. This means that all the math and trig functions behave the same as a Y-is-up CCW system, since these differences "cancel out". You can think of rotations in both systems being "from X to Y".

In order to perform a rotation of 0.5 radians (about 28.65 degrees), we plug in a value of 0.5 to the formula above and evaluate to find what the actual values should be:


Rotation of 0.5 rad ≈ 28.65°
cos(0.5) ≈ 0.8775825619
sin(0.5) ≈ 0.4794255386
X (0.8776, 0.4794)
Y (-0.4794, 0.8776)
X
Y
M=
cos(0.5) -sin(0.5)
sin(0.5) cos(0.5)

Here's how that would be done in code (place the script on a Node2D):

GDScript

```
var rot = 0.5 # The rotation to apply.
var t = Transform2D()
t.x.x = cos(rot)
t.y.y = cos(rot)
t.x.y = sin(rot)
t.y.x = -sin(rot)
transform = t # Change the node's transform to what we
calculated.
```

C#

```
float rot = 0.5f; // The rotation to apply.
Transform2D t = Transform2D.Identity;
t.X.X = t.Y.Y = Mathf.Cos(rot);
t.X.Y = t.Y.X = Mathf.Sin(rot);
t.Y.X *= -1;
Transform = t; // Change the node's transform to what we
calculated.
```

To calculate the object's rotation from an existing transformation matrix, you can use `atan2(t.x.y, t.x.x)`, where t is the Transform2D.

Note

In actual projects, you can use the `rotated()` method to perform rotations.

Basis of the transformation matrix

So far we have only been working with the `x` and `y`, vectors, which are in charge of representing rotation, scale, and/or shearing (advanced, covered at the end). The X and Y vectors are together called the *basis* of the transformation matrix. The terms "basis" and "basis vectors" are important to know.

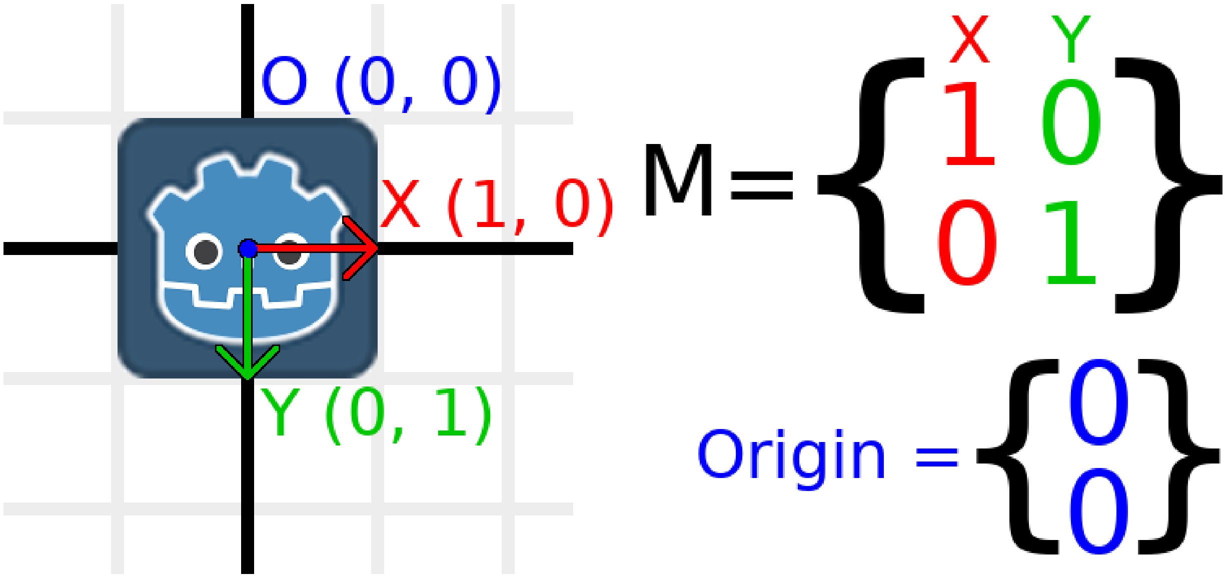
You might have noticed that Transform2D actually has three Vector2 values: `x`, `y`, and `origin`. The `origin` value is not part of the basis, but it is part of the transform, and we need it to represent position. From now on we'll keep track of the origin vector in all examples. You can think of origin as another column, but it's often better to think of it as completely separate.

Note that in 3D, Godot has a separate Basis structure for holding the three Vector3 values of the basis, since the code can get complex and it makes sense to separate it from Transform3D (which is composed of one Basis and one extra Vector3 for the origin).

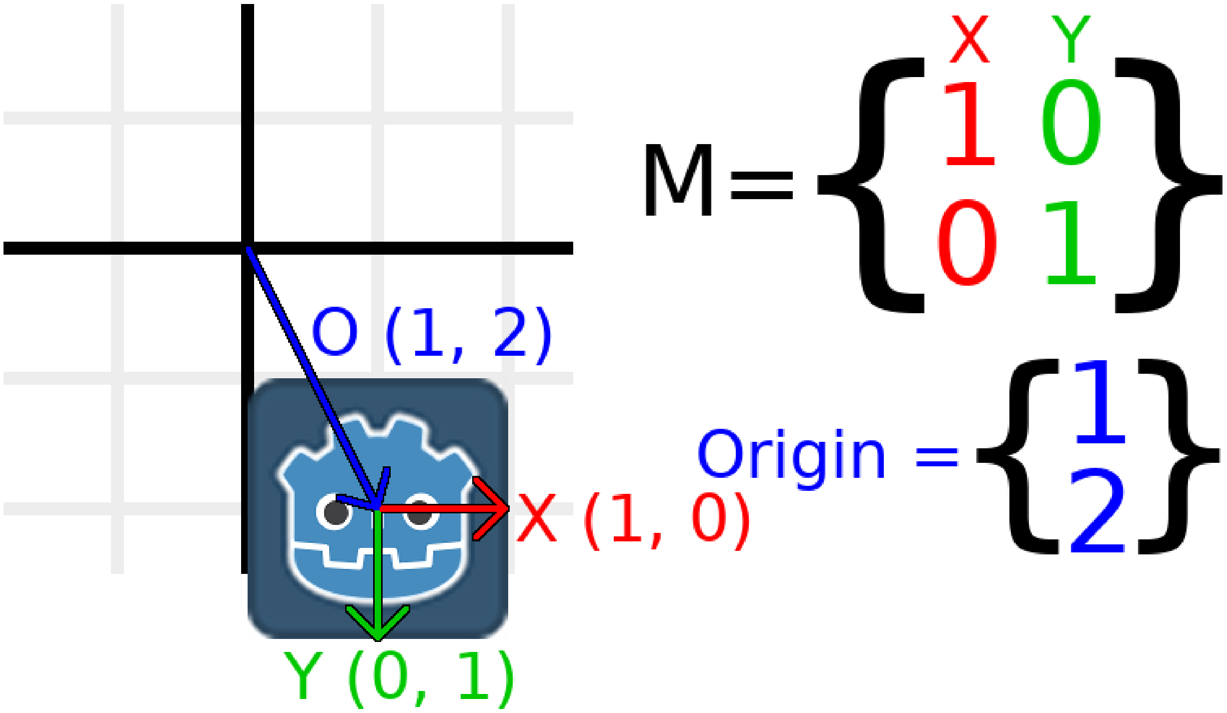
Translating the transformation matrix

Changing the `origin` vector is called *translating* the transformation matrix. Translating is basically a technical term for "moving" the object, but it explicitly does not involve any rotation.

Let's work through an example to help understand this. We will start with the identity transform like last time, except we will keep track of the origin vector this time.


O (0, 0)
X (1, 0)
Y (0, 1)
M=
X Y
1 0
0 1
Origin =
0
0

If we want to move the object to a position of (1, 2), we need to set its `origin` vector to (1, 2):


O (1, 2)
X (1, 0)
Y (0, 1)
M=
X Y
1 0
0 1
Origin =
1
2

There is also a `translated_local()` method, which performs a different operation to adding or changing `origin` directly. The `translated_local()` method will translate the object *relative to its* *own rotation*. For example, an object rotated 90 degrees clockwise will move to the right when `translated_local()` with `Vector2.UP`. To translate *relative to the global/parent frame* use `translated()` instead.

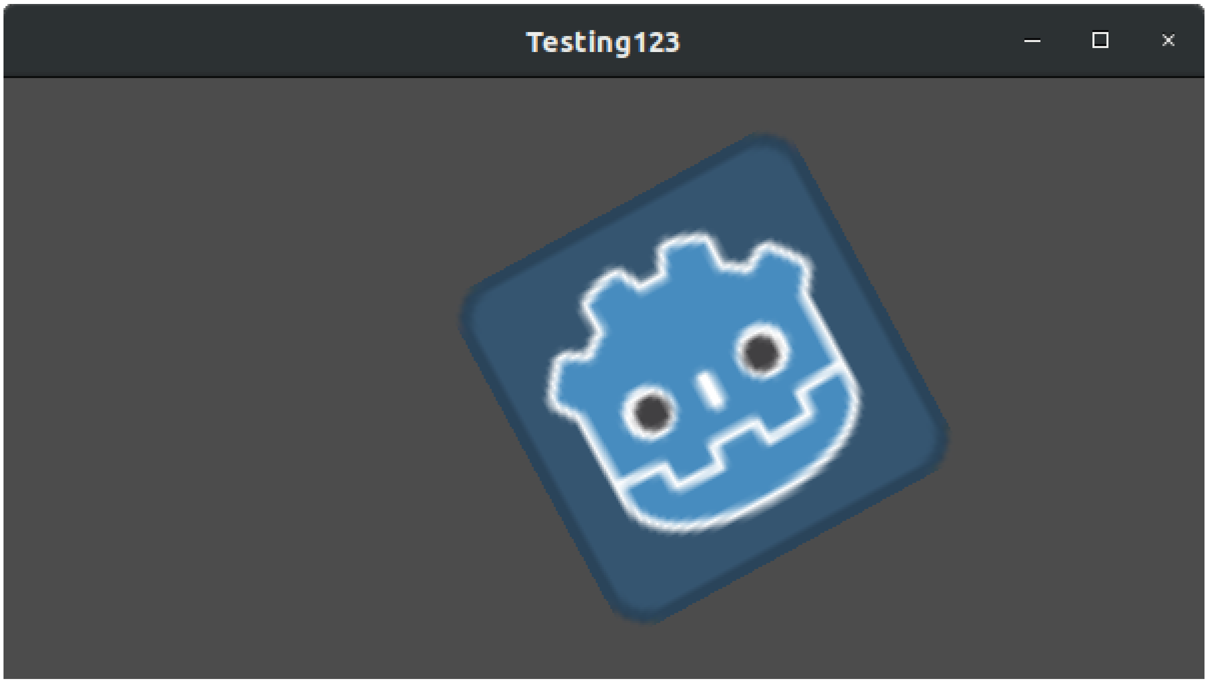
Note

Godot's 2D uses coordinates based on pixels, so in actual projects you will want to translate by hundreds of units.

Putting it all together

We're going to apply everything we mentioned so far onto one transform. To follow along, create a project with a Sprite2D node and use the Godot logo for the texture resource.

Let's set the translation to (350, 150), rotate by -0.5 rad, and scale by 3. I've posted a screenshot, and the code to reproduce it, but I encourage you to try and reproduce the screenshot without looking at the code!



GDScript

```
var t = Transform2D()
# Translation
t.origin = Vector2(350, 150)
# Rotation
var rot = -0.5 # The rotation to apply.
t.x.x = cos(rot)
t.y.y = cos(rot)
t.x.y = sin(rot)
t.y.x = -sin(rot)
# Scale
t.x *= 3
t.y *= 3
transform = t # Change the node's transform to what we
calculated.
```

C#

```
Transform2D t = Transform2D.Identity;
// Translation
t.Origin = new Vector2(350, 150);
// Rotation
float rot = -0.5f; // The rotation to apply.
```

```
t.X.X = t.Y.Y = Mathf.Cos(rot);
t.X.Y = t.Y.X = Mathf.Sin(rot);
t.Y.X *= -1;
// Scale
t.X *= 3;
t.Y *= 3;
Transform = t; // Change the node's transform to what we
calculated.
```

Shearing the transformation matrix (advanced)

Note

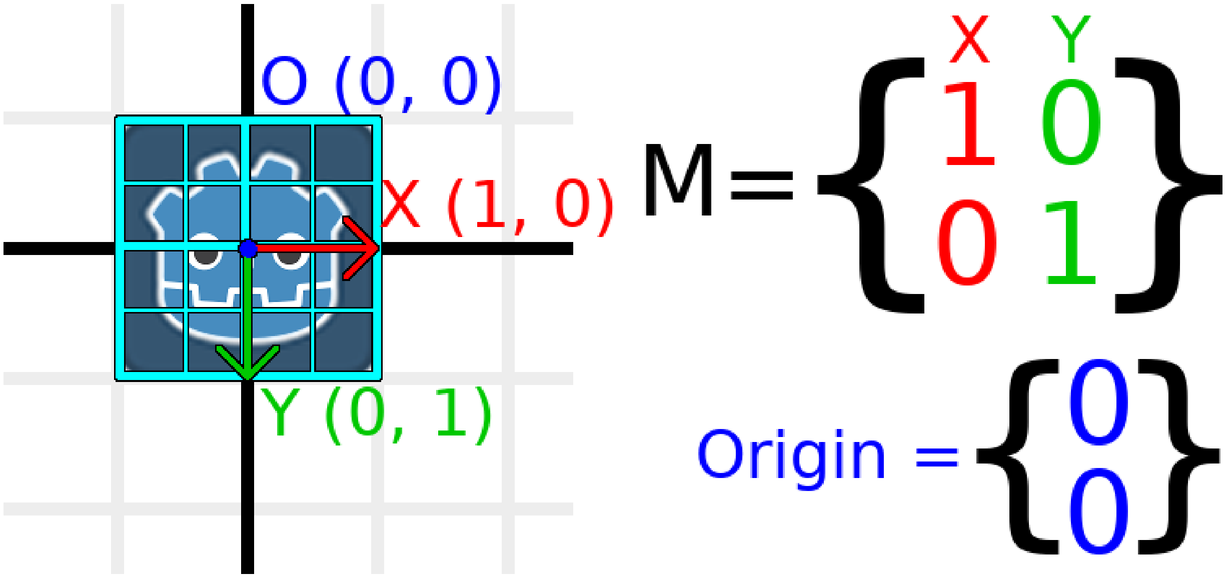
If you are only looking for how to *use* transformation matrices, feel free to skip this section of the tutorial. This section explores an uncommonly used aspect of transformation matrices for the purpose of building an understanding of them.

Node2D provides a shearing property out of the box.

You may have noticed that a transform has more degrees of freedom than the combination of the above actions. The basis of a 2D transformation matrix has four total numbers in two Vector2 values, while a rotation value and a Vector2 for scale only has 3 numbers. The high-level concept for the missing degree of freedom is called *shearing*.

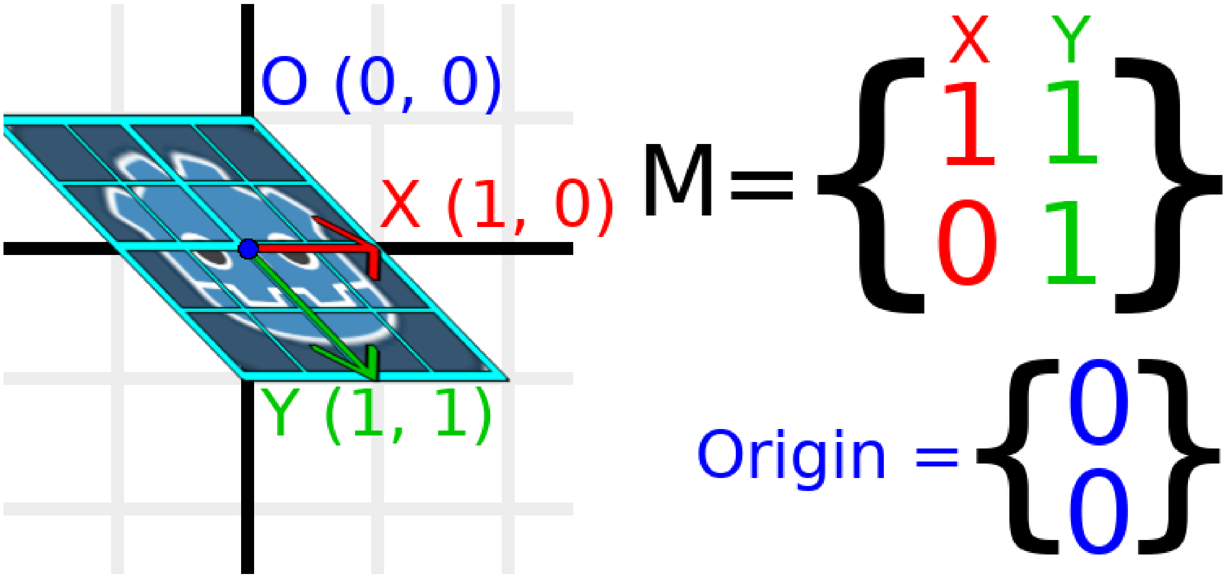
Normally, you will always have the basis vectors perpendicular to each other. However, shearing can be useful in some situations, and understanding shearing helps you understand how transforms work.

To show you visually how it will look, let's overlay a grid onto the Godot logo:


O (0, 0)
X (1, 0)
Y (0, 1)
M=
X Y
1 0
0 1
Origin =
0
0

Each point on this grid is obtained by adding the basis vectors together. The bottom-right corner is X + Y, while the top-right corner is X - Y. If we change the basis vectors, the entire grid moves with it, as the grid is composed of the basis vectors. All lines on the grid that are currently parallel will remain parallel no matter what changes we make to the basis vectors.

As an example, let's set Y to (1, 1):


O (0, 0)
X (1, 0)
Y (1, 1)
M=
X Y
1 1
0 1
Origin =
0
0

GDScript

```
var t = Transform2D()
# Shear by setting Y to (1, 1)
t.y = Vector2.ONE
transform = t # Change the node's transform to what we
calculated.
```

C#

```
Transform2D t = Transform2D.Identity;
// Shear by setting Y to (1, 1)
t.Y = Vector2.One;
Transform = t; // Change the node's transform to what we
calculated.
```

Note

You can't set the raw values of a Transform2D in the editor, so you *must* use code if you want to shear the object.

Due to the vectors no longer being perpendicular, the object has been sheared. The bottom-center of the grid, which is (0, 1) relative to itself, is now located at a world position of (1, 1).

The intra-object coordinates are called UV coordinates in textures, so let's borrow that terminology for here. To find the world position from a relative position, the formula is U * X + V * Y, where U and V are numbers and X and Y are the basis vectors.

The bottom-right corner of the grid, which is always at the UV position of (1, 1), is at the world position of (2, 1), which is calculated from X*1 + Y*1, which is (1, 0) + (1, 1), or (1 + 1, 0 + 1), or (2, 1). This matches up with our observation of where the bottom-right corner of the image is.

Similarly, the top-right corner of the grid, which is always at the UV position of (1, -1), is at the world position of (0, -1), which is calculated from X*1 + Y*-1, which is (1, 0) - (1, 1), or (1 - 1, 0 - 1), or (0, -1). This matches up with our observation of where the top-right corner of the image is.

Hopefully you now fully understand the how a transformation matrix affects the object, and the relationship between the basis vectors and how the object's "UV" or "intra-coordinates" have their world position changed.

Note

In Godot, all transform math is done relative to the parent node. When we refer to "world position", that would be relative to the node's parent instead, if the node had a parent.

If you would like additional explanation, you should check out 3Blue1Brown's excellent video about linear transformations: https://www.youtube.com/watch?v=kYB8IZa5AuE

Practical applications of transforms

In actual projects, you will usually be working with transforms inside transforms by having multiple Node2D or Node3D nodes parented to each other.

However, it's useful to understand how to manually calculate the values we need. We will go over how you could use Transform2D or Transform3D to manually calculate transforms of nodes.

Converting positions between transforms

There are many cases where you'd want to convert a position in and out of a transform. For example, if you have a position relative to the player and would like to find the world (parent-relative) position, or if you have a world position and want to know where it is relative to the player.

We can find what a vector relative to the player would be defined in world space as using the `*` operator:

GDScript

```
# World space vector 100 units below the player.
print(transform * Vector2(0, 100))
```

C#

```
// World space vector 100 units below the player.
GD.Print(Transform * new Vector2(0, 100));
```

And we can use the `*` operator in the opposite order to find a what world space position would be if it was defined relative to the player:

GDScript

```
# Where is (0, 100) relative to the player?
print(Vector2(0, 100) * transform)
```

C#

```
// Where is (0, 100) relative to the player?
GD.Print(new Vector2(0, 100) * Transform);
```

Note

If you know in advance that the transform is positioned at (0, 0), you can use the "basis_xform" or "basis_xform_inv" methods instead, which skip dealing with translation.

Moving an object relative to itself

A common operation, especially in 3D games, is to move an object relative to itself. For example, in first-person shooter games, you would want the character to move forward (-Z axis) when you press `W`.

Since the basis vectors are the orientation relative to the parent, and the origin vector is the position relative to the parent, we can add multiples of the basis vectors to move an object relative to itself.

This code moves an object 100 units to its own right:

GDScript

```
transform.origin += transform.x * 100
```

C#

```
Transform2D t = Transform;
t.Origin += t.X * 100;
Transform = t;
```

For moving in 3D, you would need to replace "x" with "basis.x".

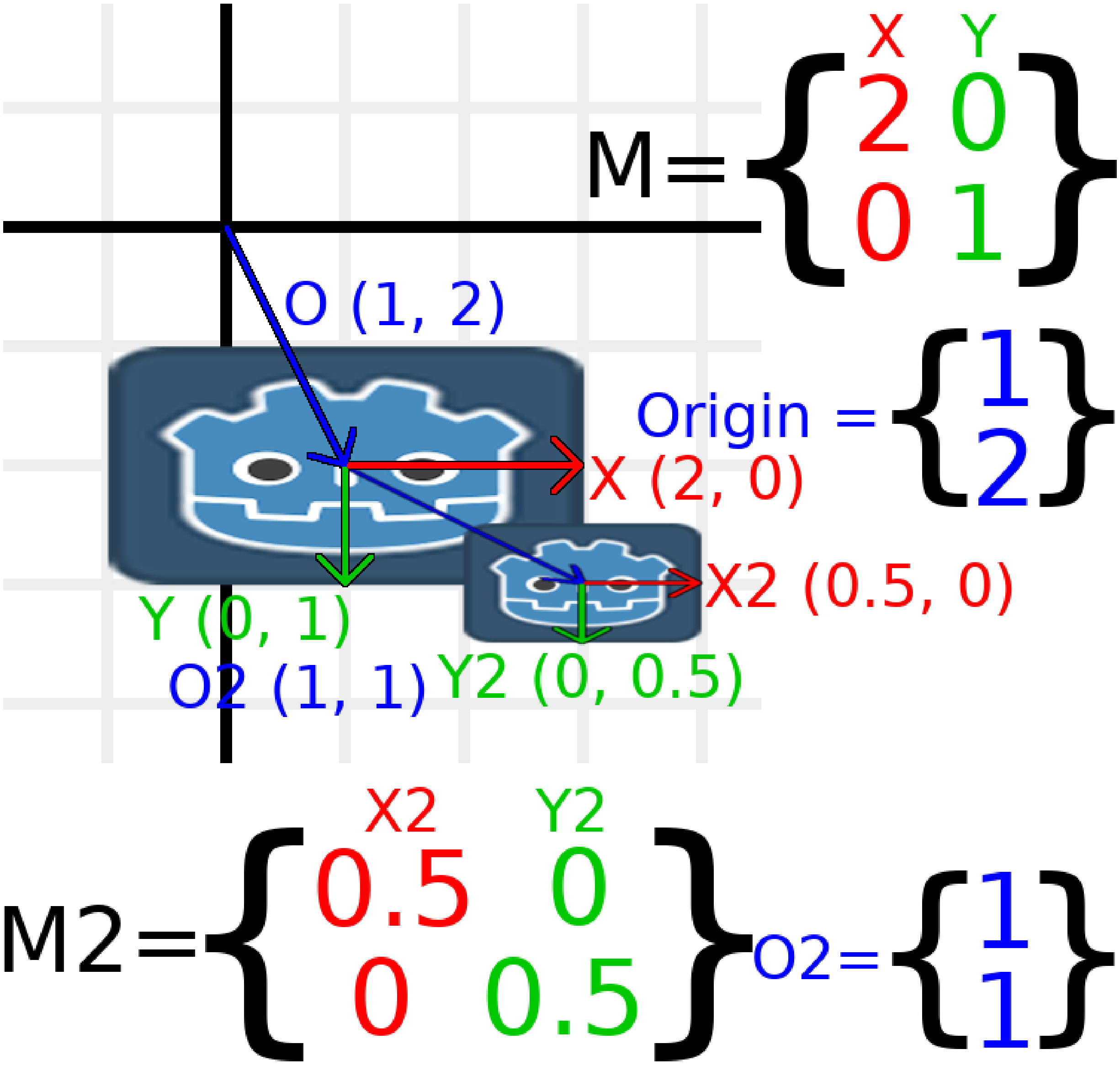
Note

In actual projects, you can use `translate_object_local` in 3D or `move_local_x` and `move_local_y` in 2D to do this.

Applying transforms onto transforms

One of the most important things to know about transforms is how you can use several of them together. A parent node's transform affects all of its children. Let's dissect an example.

In this image, the child node has a "2" after the component names to distinguish them from the parent node. It might look a bit overwhelming with so many numbers, but remember that each number is displayed twice (next to the arrows and also in the matrices), and that almost half of the numbers are zero.



X
Y
M=
2 0
0 1
O (1, 2)
Origin =
1
2
X (2, 0)
X2 (0.5, 0)
Y (0, 1)
O2 (1, 1)
Y2 (0, 0.5)
X2
Y2
M2=
0.5 0
0 0.5
O2=
1
1

The only transformations going on here are that the parent node has been given a scale of (2, 1), the child has been given a scale of (0.5, 0.5), and both nodes have been given positions.

All child transformations are affected by the parent transformations. The child has a scale of (0.5, 0.5), so you would expect it to be a 1:1 ratio square, and it is, but only relative to the parent. The child's X vector ends up being (1, 0) in world space, because it is scaled by the parent's basis vectors. Similarly, the child node's `origin` vector is set to (1, 1), but this actually moves it (2, 1) in world space, due to the parent node's basis vectors.

To calculate a child transform's world space transform manually, this is the code we would use:

GDScript

```
# Set up transforms like in the image, except make positions be
100 times bigger.
var parent = Transform2D(Vector2(2, 0), Vector2(0, 1),
Vector2(100, 200))
var child = Transform2D(Vector2(0.5, 0), Vector2(0, 0.5),
Vector2(100, 100))

# Calculate the child's world space transform
# origin = (2, 0) * 100 + (0, 1) * 100 + (100, 200)
var origin = parent.x * child.origin.x + parent.y *
child.origin.y + parent.origin
# basis_x = (2, 0) * 0.5 + (0, 1) * 0
var basis_x = parent.x * child.x.x + parent.y * child.x.y
# basis_y = (2, 0) * 0 + (0, 1) * 0.5
var basis_y = parent.x * child.y.x + parent.y * child.y.y

# Change the node's transform to what we calculated.
transform = Transform2D(basis_x, basis_y, origin)
```

C#

```
// Set up transforms like in the image, except make positions
be 100 times bigger.
Transform2D parent = new Transform2D(2, 0, 0, 1, 100, 200);
Transform2D child = new Transform2D(0.5f, 0, 0, 0.5f, 100,
100);
```

```
// Calculate the child's world space transform
// origin = (2, 0) * 100 + (0, 1) * 100 + (100, 200)
Vector2 origin = parent.X * child.Origin.X + parent.Y *
child.Origin.Y + parent.Origin;
// basisX = (2, 0) * 0.5 + (0, 1) * 0 = (0.5, 0)
Vector2 basisX = parent.X * child.X.X + parent.Y * child.X.Y;
// basisY = (2, 0) * 0 + (0, 1) * 0.5 = (0.5, 0)
Vector2 basisY = parent.X * child.Y.X + parent.Y * child.Y.Y;

// Change the node's transform to what we calculated.
Transform = new Transform2D(basisX, basisY, origin);
```

In actual projects, we can find the world transform of the child by applying one transform onto another using the `*` operator:

GDScript

```
# Set up transforms like in the image, except make positions be
100 times bigger.
var parent = Transform2D(Vector2(2, 0), Vector2(0, 1),
Vector2(100, 200))
var child = Transform2D(Vector2(0.5, 0), Vector2(0, 0.5),
Vector2(100, 100))

# Change the node's transform to what would be the child's
world transform.
transform = parent * child
```

C#

```
// Set up transforms like in the image, except make positions
be 100 times bigger.
Transform2D parent = new Transform2D(2, 0, 0, 1, 100, 200);
Transform2D child = new Transform2D(0.5f, 0, 0, 0.5f, 100,
100);

// Change the node's transform to what would be the child's
world transform.
Transform = parent * child;
```

Note

When multiplying matrices, order matters! Don't mix them up.

Lastly, applying the identity transform will always do nothing.

If you would like additional explanation, you should check out 3Blue1Brown's excellent video about matrix composition: https://www.youtube.com/watch?v=XkY2DOUCWMU

Inverting a transformation matrix

The "affine_inverse" function returns a transform that "undoes" the previous transform. This can be useful in some situations. Let's take a look at a few examples.

Multiplying an inverse transform by the normal transform undoes all transformations:

GDScript

```
var ti = transform.affine_inverse()
var t = ti * transform
# The transform is the identity transform.
```

C#

```
Transform2D ti = Transform.AffineInverse();
Transform2D t = ti * Transform;
// The transform is the identity transform.
```

Transforming a position by a transform and its inverse results in the same position:

GDScript

```
var ti = transform.affine_inverse()
position = transform * position
position = ti * position
# The position is the same as before.
```

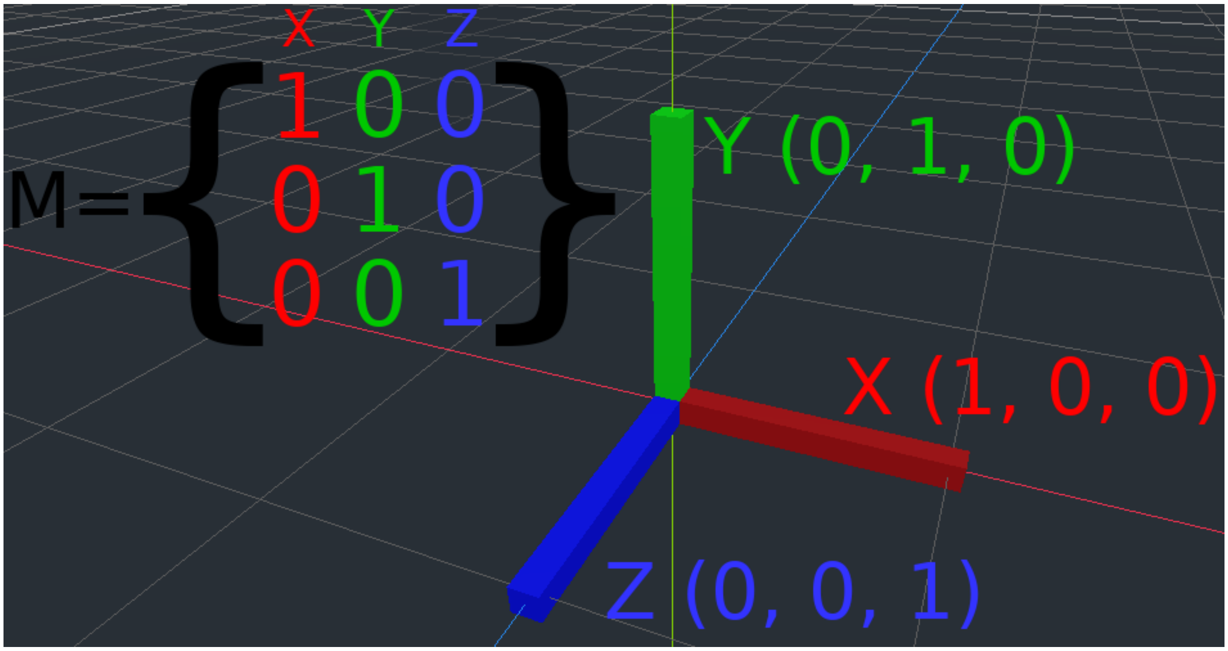
C#

```
Transform2D ti = Transform.AffineInverse();
Position = Transform * Position;
Position = ti * Position;
// The position is the same as before.
```

How does it all work in 3D?

One of the great things about transformation matrices is that they work very similarly between 2D and 3D transformations. All the code and formulas used above for 2D work the same in 3D, with 3 exceptions: the addition of a third axis, that each axis is of type Vector3, and also that Godot stores the Basis separately from the Transform3D, since the math can get complex and it makes sense to separate it.

All of the concepts for how translation, rotation, scale, and shearing work in 3D are all the same compared to 2D. To scale, we take each component and multiply it; to rotate, we change where each basis vector is pointing; to translate, we manipulate the origin; and to shear, we change the basis vectors to be non-perpendicular.



If you would like, it's a good idea to play around with transforms to get an understanding of how they work. Godot allows you to edit 3D transform matrices directly from the inspector. You can download this project which has colored lines and cubes to help visualize the Basis vectors and the origin in both 2D and 3D: https://github.com/godotengine/godot-demo-projects/tree/master/misc/matrix_transform

Note

You cannot edit Node2D's transform matrix directly in Godot 4.0's inspector. This may be changed in a future release of Godot.

If you would like additional explanation, you should check out 3Blue1Brown's excellent video about 3D linear transformations: https://www.youtube.com/watch?v=rHLEWRxRGiM

Representing rotation in 3D (advanced)

The biggest difference between 2D and 3D transformation matrices is how you represent rotation by itself without the basis vectors.

With 2D, we have an easy way (atan2) to switch between a transformation matrix and an angle. In 3D, rotation is too complex to represent as one number. There is something called Euler angles, which can represent rotations as a set of 3 numbers, however, they are limited and not very useful, except for trivial cases.

In 3D we do not typically use angles, we either use a transformation basis (used pretty much everywhere in Godot), or we use quaternions. Godot can represent quaternions using the Quaternion struct. My suggestion to you is to completely ignore how they work under-the-hood, because they are very complicated and unintuitive.

However, if you really must know how it works, here are some great resources, which you can follow in order:

https://www.youtube.com/watch?v=mvmuCPvRoWQ

https://www.youtube.com/watch?v=d4EgbgTm0Bg

https://eater.net/quaternions

Interpolation

Interpolation is a very basic operation in graphics programming. It's good to become familiar with it in order to expand your horizons as a graphics developer.

The basic idea is that you want to transition from A to B. A value `t`, represents the states in-between.

For example, if `t` is 0, then the state is A. If `t` is 1, then the state is B. Anything in-between is an *interpolation*.

Between two real (floating-point) numbers, an interpolation can be described as:

```
interpolation = A * (1 - t) + B * t
```

And often simplified to:

```
interpolation = A + (B - A) * t
```

The name of this type of interpolation, which transforms a value into another at *constant speed* is *"linear"*. So, when you hear about *Linear Interpolation*, you know they are referring to this formula.

There are other types of interpolations, which will not be covered here. A recommended read afterwards is the Bezier page.

Vector interpolation

Vector types (Vector2 and Vector3) can also be interpolated, they come with handy functions to do it Vector2.lerp() and Vector3.lerp().

For cubic interpolation, there are also Vector2.cubic_interpolate() and Vector3.cubic_interpolate(), which do a Bezier style interpolation.

Here is example pseudo-code for going from point A to B using interpolation:

GDScript

```
var t = 0.0

func _physics_process(delta):
    t += delta * 0.4

    $Sprite2D.position = $A.position.lerp($B.position, t)
```

C#

```
private float _t = 0.0f;

public override void _PhysicsProcess(double delta)
{
    _t += (float)delta * 0.4f;

    Marker2D a = GetNode<Marker2D>("A");
    Marker2D b = GetNode<Marker2D>("B");
    Sprite2D sprite = GetNode<Sprite2D>("Sprite2D");

    sprite.Position = a.Position.Lerp(b.Position, _t);
}
```

It will produce the following motion:

Transform interpolation

It is also possible to interpolate whole transforms (make sure they have either uniform scale or, at least, the same non-uniform scale). For this, the function Transform3D.interpolate_with() can be used.

Here is an example of transforming a monkey from Position1 to Position2:

Position2
Position1

Using the following pseudocode:

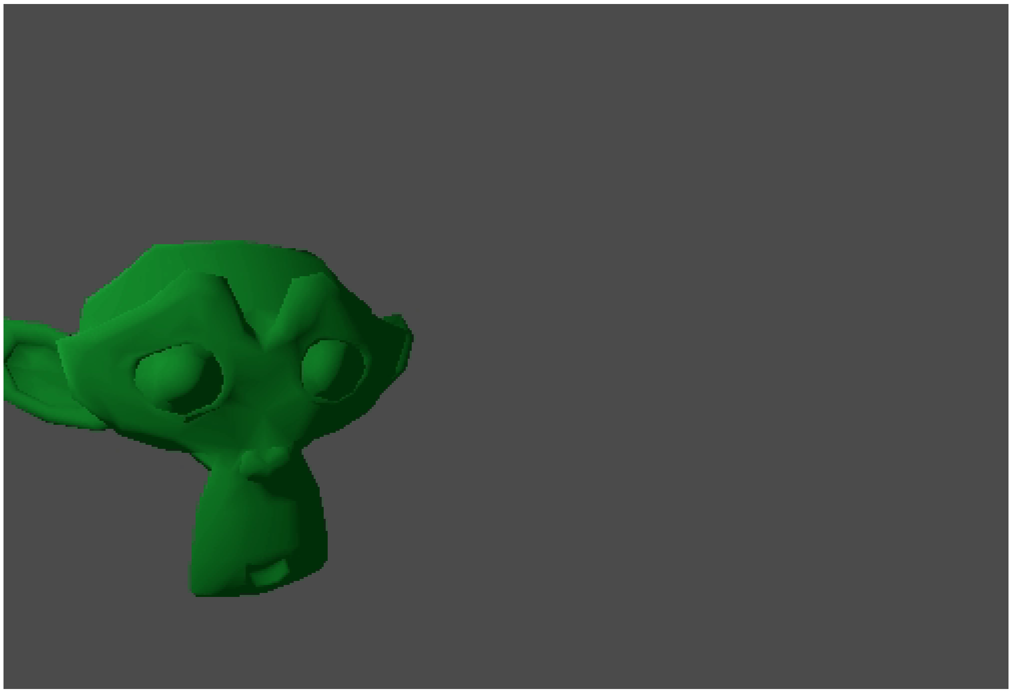
GDScript

```
var t = 0.0

func _physics_process(delta):
    t += delta

    $Monkey.transform =
$Position1.transform.interpolate_with($Position2.transform, t)
```

C#

```
private float _t = 0.0f;

public override void _PhysicsProcess(double delta)
{
    _t += (float)delta;

    Marker3D p1 = GetNode<Marker3D>("Position1");
    Marker3D p2 = GetNode<Marker3D>("Position2");
    CSGMesh3D monkey = GetNode<CSGMesh3D>("Monkey");

    monkey.Transform =
p1.Transform.InterpolateWith(p2.Transform, _t);
}
```

And again, it will produce the following motion:



Smoothing motion

Interpolation can be used to smooth movement, rotation, etc. Here is an example of a circle following the mouse using smoothed motion:

GDScript

```
const FOLLOW_SPEED = 4.0

func _physics_process(delta):
    var mouse_pos = get_local_mouse_position()

    $Sprite2D.position = $Sprite2D.position.lerp(mouse_pos,
delta * FOLLOW_SPEED)
```

C#

```
private const float FollowSpeed = 4.0f;

public override void _PhysicsProcess(double delta)
{
    Vector2 mousePos = GetLocalMousePosition();

    Sprite2D sprite = GetNode<Sprite2D>("Sprite2D");

    sprite.Position = sprite.Position.Lerp(mousePos,
(float)delta * FollowSpeed);
}
```

Here is how it looks:

This is useful for smoothing camera movement, allies following you (ensuring they stay within a certain range), and many other common game patterns.

Beziers, curves and paths

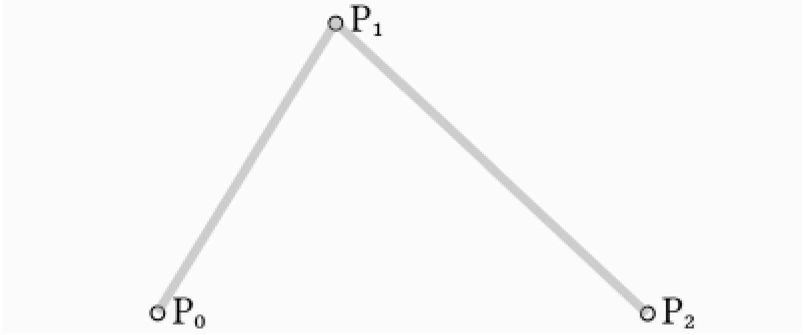
Bezier curves are a mathematical approximation of natural geometric shapes. We use them to represent a curve with as little information as possible and with a high level of flexibility.

Unlike more abstract mathematical concepts, Bezier curves were created for industrial design. They are a popular tool in the graphics software industry.

They rely on interpolation, which we saw in the previous article, combining multiple steps to create smooth curves. To better understand how Bezier curves work, let's start from its simplest form: Quadratic Bezier.

Quadratic Bezier

Take three points, the minimum required for Quadratic Bezier to work:



To draw a curve between them, we first interpolate gradually over the two vertices of each of the two segments formed by the three points, using values ranging from 0 to 1. This gives us two points that move along the segments as we change the value of `t` from 0 to 1.

GDScript

```
func _quadratic_bezier(p0: Vector2, p1: Vector2, p2: Vector2,
t: float):
    var q0 = p0.lerp(p1, t)
    var q1 = p1.lerp(p2, t)
```

C#

```
private Vector2 QuadraticBezier(Vector2 p0, Vector2 p1, Vector2
p2, float t)
{
    Vector2 q0 = p0.Lerp(p1, t);
    Vector2 q1 = p1.Lerp(p2, t);
}
```

We then interpolate `q0` and `q1` to obtain a single point `r` that moves along a curve.

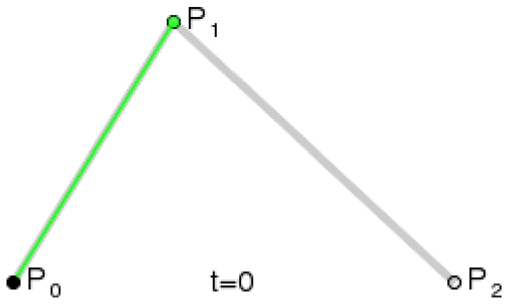
GDScript

```
var r = q0.lerp(q1, t)
return r
```

C#

```
Vector2 r = q0.Lerp(q1, t);
return r;
```

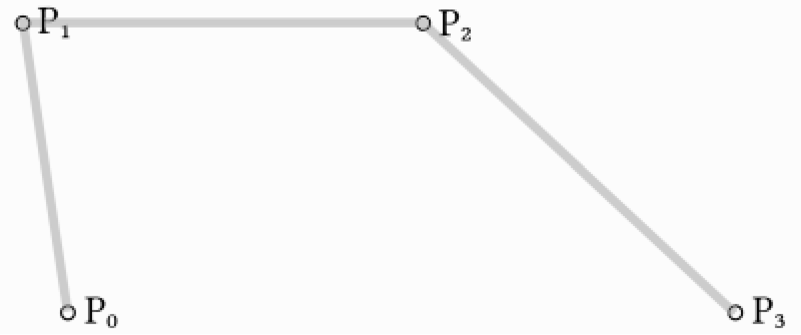
This type of curve is called a *Quadratic Bezier* curve.



(Image credit: Wikipedia)

Cubic Bezier

Building upon the previous example, we can get more control by interpolating between four points.



We first use a function with four parameters to take four points as an input, `p0`, `p1`, `p2` and `p3`:

GDScript

```
func _cubic_bezier(p0: Vector2, p1: Vector2, p2: Vector2, p3:
Vector2, t: float):
```

C#

```
public Vector2 CubicBezier(Vector2 p0, Vector2 p1, Vector2 p2,
Vector2 p3, float t)
{

}
```

We apply a linear interpolation to each couple of points to reduce them to three:

GDScript

```
var q0 = p0.lerp(p1, t)
var q1 = p1.lerp(p2, t)
var q2 = p2.lerp(p3, t)
```

C#

```
Vector2 q0 = p0.Lerp(p1, t);
Vector2 q1 = p1.Lerp(p2, t);
Vector2 q2 = p2.Lerp(p3, t);
```

We then take our three points and reduce them to two:

GDScript

```
var r0 = q0.lerp(q1, t)
var r1 = q1.lerp(q2, t)
```

C#

```
Vector2 r0 = q0.Lerp(q1, t);
Vector2 r1 = q1.Lerp(q2, t);
```

And to one:

GDScript

```
var s = r0.lerp(r1, t)
return s
```

C#

```
Vector2 s = r0.Lerp(r1, t);
return s;
```

Here is the full function:

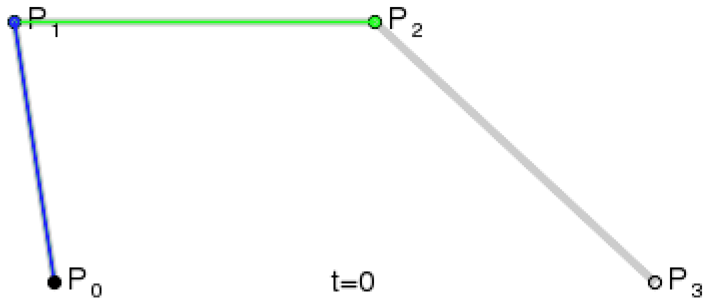
GDScript

```
func _cubic_bezier(p0: Vector2, p1: Vector2, p2: Vector2, p3:
Vector2, t: float):
    var q0 = p0.lerp(p1, t)
    var q1 = p1.lerp(p2, t)
    var q2 = p2.lerp(p3, t)

    var r0 = q0.lerp(q1, t)
    var r1 = q1.lerp(q2, t)

    var s = r0.lerp(r1, t)
    return s
```

C#

```
private Vector2 CubicBezier(Vector2 p0, Vector2 p1, Vector2 p2,
Vector2 p3, float t)
{
    Vector2 q0 = p0.Lerp(p1, t);
    Vector2 q1 = p1.Lerp(p2, t);
    Vector2 q2 = p2.Lerp(p3, t);

    Vector2 r0 = q0.Lerp(q1, t);
    Vector2 r1 = q1.Lerp(q2, t);

    Vector2 s = r0.Lerp(r1, t);
    return s;
}
```

The result will be a smooth curve interpolating between all four points:



(Image credit: Wikipedia)

Note

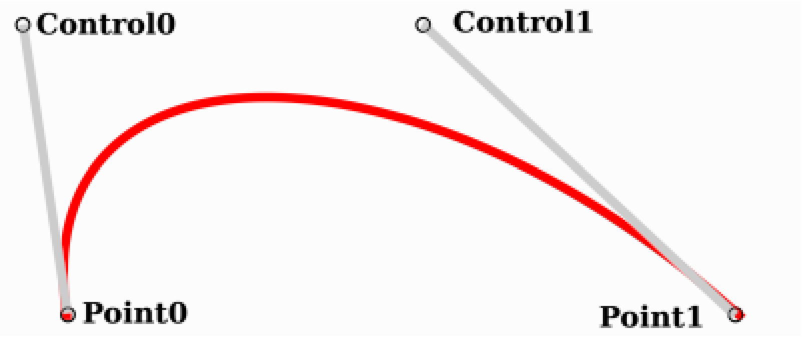
Cubic Bezier interpolation works the same in 3D, just use `Vector3` instead of `Vector2`.

Adding control points

Building upon Cubic Bezier, we can change the way two of the points work to control the shape of our curve freely. Instead of having `p0`, `p1`, `p2` and `p3`, we will store them as:

- `point0 = p0`: Is the first point, the source
- `control0 = p1 - p0`: Is a vector relative to the first control point
- `control1 = p3 - p2`: Is a vector relative to the second control point
- `point1 = p3`: Is the second point, the destination

This way, we have two points and two control points which are relative vectors to the respective points. If you've used graphics or animation software before, this might look familiar:



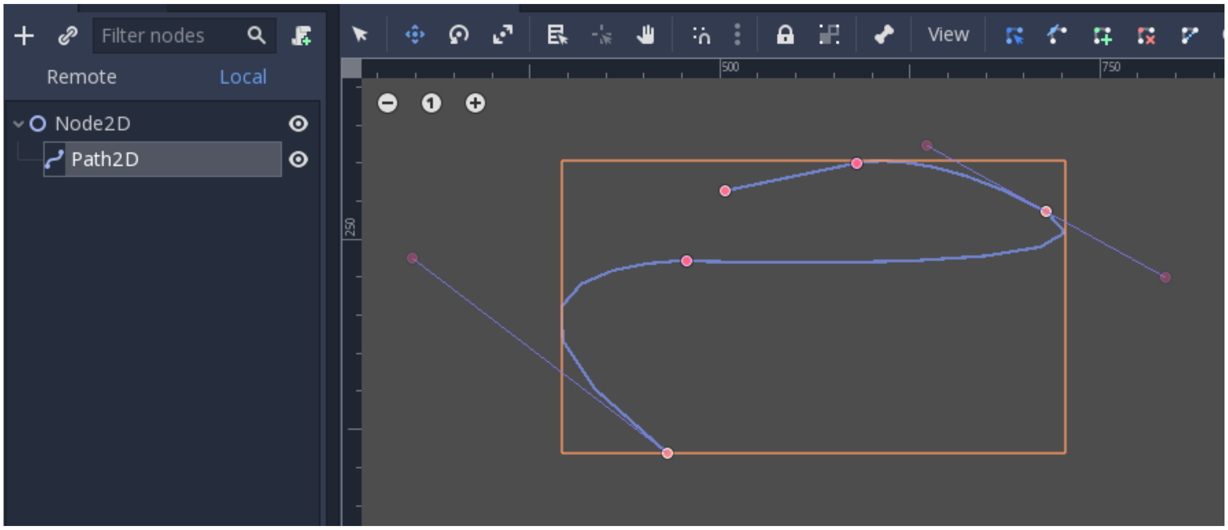
This is how graphics software presents Bezier curves to the users, and how they work and look in Godot.

Curve2D, Curve3D, Path and Path2D

There are two objects that contain curves: Curve3D and Curve2D (for 3D and 2D respectively).

They can contain several points, allowing for longer paths. It is also possible to set them to nodes: Path3D and Path2D (also for 3D and

2D respectively):



Using them, however, may not be completely obvious, so following is a description of the most common use cases for Bezier curves.

Evaluating

Only evaluating them may be an option, but in most cases it's not very useful. The big drawback with Bezier curves is that if you traverse them at constant speed, from `t = 0` to `t = 1`, the actual interpolation will *not* move at constant speed. The speed is also an interpolation between the distances between points `p0`, `p1`, `p2` and `p3` and there is not a mathematically simple way to traverse the curve at constant speed.

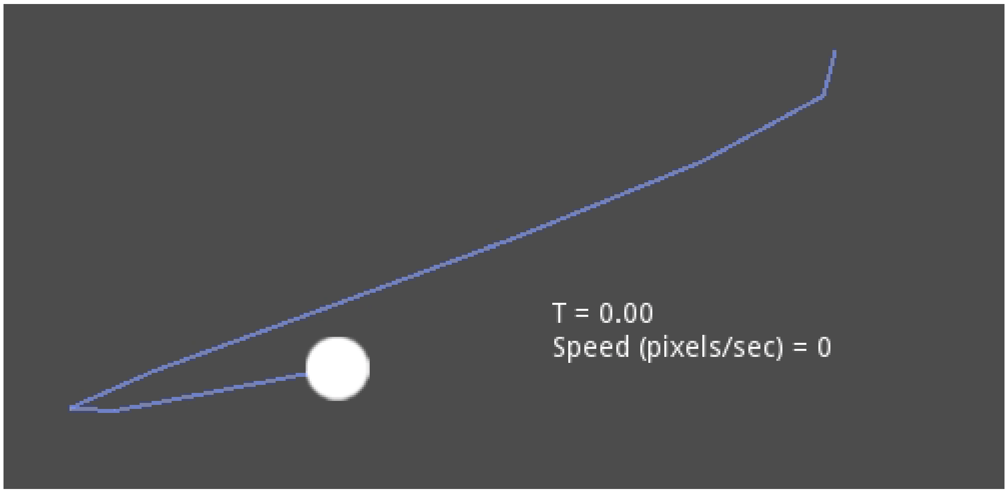
Let's do an example with the following pseudocode:

GDScript

```
var t = 0.0

func _process(delta):
    t += delta
    position = _cubic_bezier(p0, p1, p2, p3, t)
```

C#

```
private float _t = 0.0f;

public override void _Process(double delta)
{
    _t += (float)delta;
    Position = CubicBezier(p0, p1, p2, p3, _t);
}
```

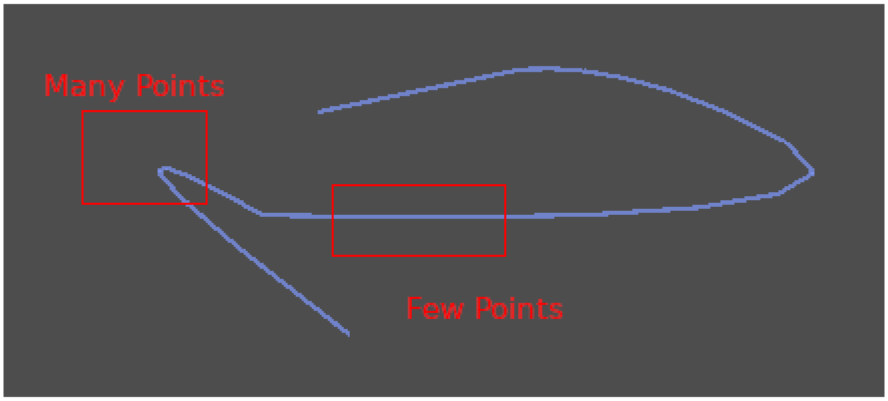


As you can see, the speed (in pixels per second) of the circle varies, even though `t` is increased at constant speed. This makes beziers difficult to use for anything practical out of the box.

Drawing

Drawing beziers (or objects based on the curve) is a very common use case, but it's also not easy. For pretty much any case, Bezier curves need to be converted to some sort of segments. This is normally difficult, however, without creating a very high amount of them.

The reason is that some sections of a curve (specifically, corners) may require considerable amounts of points, while other sections may not:



Additionally, if both control points were `0, 0` (remember they are relative vectors), the Bezier curve would just be a straight line (so drawing a high amount of points would be wasteful).

Before drawing Bezier curves, *tessellation* is required. This is often done with a recursive or divide and conquer function that splits the curve until the curvature amount becomes less than a certain threshold.

The *Curve* classes provide this via the Curve2D.tessellate() function (which receives optional `stages` of recursion and angle `tolerance` arguments). This way, drawing something based on a curve is easier.

Traversal

The last common use case for the curves is to traverse them. Because of what was mentioned before regarding constant speed, this is also difficult.

To make this easier, the curves need to be *baked* into equidistant points. This way, they can be approximated with regular interpolation (which can be improved further with a cubic option). To